PATENT
20002/16136

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant:

Mingqiu Sun

Serial No.: 10/424,356

For: METHODS AND APPARATUS TO DETECT PATTERNS IN PROGRAMS

Filed: April 28 2003

Group Art Unit: 2192

Examiner: John J. Romano

DECLARATION OF JAMES A. FLIGHT

Mail Stop Amendment
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

I, James A. Flight, hereby declare and state:

1. I am attorney of record for the above-referenced patent application.

2. The United States Patent & Trademark Office issued an Office action dated March 21, 2006, purporting to reject all pending claims of the above-referenced patent application based on Sherwood et al., Phase Tracking and Prediction, technical report CS2002-0710, UC San Diego, June 2002. That technical report can be found on the Internet at http://www.cse.ucsd.edu/Dienst/UI/2.0/Describe/ncstrl.ucsd_cse/CS2002-0710. A copy of the technical report is attached hereto as Exhibit 1.

3. As shown in Exhibit 1, the technical report includes two links. Both of those links provide the ability to download a 10Kb postscript file. A copy of that postscript file is attached hereto as Exhibit 2.

4. Exhibit 2 states "To obtain a copy of this techreport, please look for it at the following site:

http://www-cse.ucsd.edu/users/calder/papers.html

Or send email or letter to:

Brad Calder... calder@cs.ucsd.edu" Following the link in Exhibit 2 leads to Exhibit 3.

5. Exhibit 3 does ***not*** list the technical report as published in 2002. However, it identifies a corresponding paper as published in ***June of 2003***, namely, Timothy Sherwood, Suleyman Sair, and Brad Calder, **Phase Tracking and Prediction,** 30th International Symposium on Computer Architecture, June 2003. The PDF of this June of 2003 paper is linked in Exhibit 3. A copy of this June of 2003 paper is attached hereto as Exhibit 4. Exhibit 4 identifies its date of publication as June of 2003.

6. The Office action of March 21, 2006 purports to reject the claims noted in paragraph 2 above based on the technical report of 2002. However, *it actually relies on the content of the June of 2003 publication* (i.e., Exhibit 4) to support the rejections. As noted above, the technical report of 2002 (Exhibit 1) includes only an abstract. It does not include Exhibit 4.

7. Because of the evident inconsistency between Exhibit 1 and Exhibit 4, I sent an email to Timothy Sherwood, the first listed author in

Exhibits 1 and 4 on October 30, 2006 asking Mr. Sherwood to identify the correct date of publication for the full article. As shown in Exhibit 5, Mr. Sherwood replied to my email on October 30, 2006 by stating:

> Hi Jim,
>
> The publication appeared in ISCA 2003 and so was officially published on June 9th, 2003.
>
> http://cs.nyu.edu/isca03/
>
> -Tim

8. The citation in Mr. Sherwood's email (Exhibit 5) is to the program announcement shown in Exhibit 6. That program announcement states "The thirtieth International Symposium on Computer Architecture (ISCA) will be held at the Town and Country Hotel in San Diego 9-11 June, 2003." Thus, Mr. Sherwood, the first named author of Exhibits 1 and 4 indicated that the full publication was made available in June of 2003, ***which is after the April 28, 2003 filing date of the instant application.***

9. Having no reason whatsoever to doubt Mr. Sherwood's testimony, I prepared and filed a response to the Office action dated March 21, 2006 attaching a copy of Mr. Sherwood's email and explaining that, based on the author's testimony, the full Sherwood article (i.e., Exhibit 4) is not prior art to the instant application.

10. The USPTO issued a final Office action on May 16, 2007 refusing to accept the author's testimony because it was not in the form of an affidavit or declaration. Accordingly, I am submitting this requested declaration to verify the source of the emails noted above.

11. The final Office action attempts to locate evidence that the author, Mr. Sherwood, is incorrect in his belief that the publication of Exhibit 4

did not occur until after the filing date of the instant application. In particular, the final Office action cites four pieces of evidence to allegedly support an earlier publication date for the full article.

12. First, the final Office action cites the publication "ACM SIGARCH Computer Architecture News, archive volume 31, Issue 2 (May 2003), pages 336-349, ISSN:0163-5964" (attached hereto as Exhibit 7) as evidencing publication prior to June of 2003. Exhibit 7 does indeed note that the Sherwood article was published in May of 2003. However, ***May of 2003 is still after the April 28, 2003 filing date of this application.*** Therefore, Exhibit 7 ***fails*** to make the full Sherwood article (Exhibit 4) prior art to this application. A copy of the Sherwood article as linked to in Exhibit 7 is attached hereto as Exhibit 8.

13. The final Office action also cites to Exhibits 9 and 10 (reports by year and author, respectively) which identify the date of the technical report (i.e., Exhibit 1) as June 23, 2002. However, as noted above, the technical report *is not the full article* (i.e., it is not Exhibits 4 or 8) and, thus, Exhibits 9 and 10 only serve to document Exhibit 1 as prior art, not the full article (i.e., not Exhibits 4 or 8).

14. Finally, the Office action cites to footnote 20 of "Automatically, Characterizing Large Scale Program Behavior," 2002, ISBN 1-58113-574-2 (Exhibit 11) as evidence of the availability of the technical report (i.e., Exhibit 1). Again, however, Exhibit 1 is not Exhibits 4 or 8, and, thus, the demonstration of the 2002 publication of the technical report does not make Exhibits 4 or 8 prior art. Thus, the evidence of record relied upon by the final Office action does not demonstrate Exhibits 4 and/or 8 to be prior

art to the instant application. Accordingly, the final Office action's reliance on the content of Exhibits 4 and/or 8 to reject the claims is improper.

15. To attempt to determine the earliest publication date of the full article (i.e., Exhibits 4 and/or 8), the undersigned searched the Internet Archive Wayback Machine (http://www.archive.org/index.php) for the technical report (i.e., Exhibit 1) . As shown in Exhibit 12, that search demonstrated that only an abstract of the Sherwood article (i.e., only Exhibit 1) was present on the Internet as of November 19, 2002. In particular, as shown in Exhibit 13 attached hereto (i.e., the printout from the November 19, 2002 link of Exhibit 12), only a single paragraph ***and not the full Sherwood article (i.e., not Exhibits 4 and/or 8)*** could be accessed as of the November 19, 2002. Exhibit 14, which is the printout from the July 9, 2003 link of Exhibit 12, is identical to the November 19, 2002 information. Therefore, only Exhibit 1 has been shown to be prior art. ***There is no evidence of record that the full Sherwood article (i.e., Exhibits 4 and/or 8) is prior art to this application.***

16. As noted above, Exhibit 2 invited the public to contact Brad Calder "to obtain a copy of the techreport." Therefore, there is a ***possibility*** that Mr. Calder was providing something beyond the content of Exhibit 1 to requestors prior to the filing date of this application. Accordingly, to determine if anything more than Exhibit 1 is prior art to the instant application, I sent the email attached hereto as Exhibit 15 to Mr. Calder on July 7, 2007 asking Mr. Calder to clarify the situation. Having had no response, I again sent the email attached hereto as Exhibit 16 to Mr.

Calder. As shown at Exhibit 17, Exhibit 16 was delivered to Mr. Calder. To date, he has made no response.

17. I also sent a request for clarification to the webmaster for the server that serves Exhibit 1 to the Internet asking for clarification as to the situation and to obtain the postscript file attached hereto as Exhibit 2. As shown in Exhibit 18, the webmaster responded by re-booting the server to make Exhibit 2 available, and by referring to Exhibits 9 and 10. As discussed above, ***none*** of this shows the full article (i.e., Exhibits 4 and/or 8) to be prior art. Therefore, despite all efforts by the undersigned and the Examiner to date, nothing of record indicates that anything beyond Exhibit 1 is prior art to the instant application.

18. I understand that willful and false statements and the like are punishable by a fine and/or imprisonment under 18 U.S.C. § 1001, and that such willful false statement may jeopardize the validity of this application and any patent resulting therefrom.

Date: July 16, 2007 By: James A Flight

EXHIBIT 1
TO DECLARATION OF
JAMES A FLIGHT

Phase Tracking and Prediction

Timothy Sherwood, Suleyman Sair and Brad Calder
CS2002-0710
June 23, 2002

In a single second a modern processor can execute billions of instructions. Obtaining a bird's eye view of the behavior of a program at these speeds can be a difficult task when all that is available is cycle by cycle examination. In many programs, behavior is anything but steady state, and understanding the patterns of behavior at run-time can unlock a multitude of optimization opportunities. In this paper we present a unified profiling architecture that can efficiently capture, classify, and predict program behavior on the largest of time scales all at run-time with no support from software. By examining the proportion of instructions that were executed from different sections of code, we can find generic phases that correspond to changes in behavior across many metrics. By classifying phases generically, we avoid the need to identify phases for each optimization, and enable a unified prediction scheme that can forecast future behavior. We examine the ability of our phase tracking architecture to accurately capture the phase behavior of a program's execution with respect to its overall performance (IPC), branch prediction, cache performance, and energy, and show how phase behavior may be captured efficiently using a simple predictor.

How to view this document

- Display the **whole** document in one of the following formats.
  - PostScript 10012 bytes.
- Print or download all or selected pages.

EXHIBIT 2
TO DECLARATION OF
JAMES A FLIGHT

Techreport

To obtain a copy of this techreport, please look for it at the following site:

http://www-cse.ucsd.edu/users/calder/papers.html

Or send email or a letter to:

Brad Calder
University of California, San Diego
9500 Gilman Drive
La Jolla, CA 92093-0114
calder@cs.ucsd.edu

EXHIBIT 3
TO DECLARATION OF
JAMES A FLIGHT

Publications

- **Thesis**

Publications are listed below by year of publication. For a list of papers by category, please click here .

Book Chapters

- Brad Calder, Timothy Sherwood, Greg Hamerly, and Erez Perelman, **SimPoint: Picking Representative Samples to Guide Simulation.** Chapter 7 in the book "Performance Evaluation and Benchmarking" edited by Lizy Kurian John and Lieven Eeckhout; published by CRC Press, 2005

2007

- Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards and Brad Calder , **Automatically Classifying Benign and Harmful Data Races Using Replay Analysis** , ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2007 (pdf)

- Erez Perelman, Jeremy Lau, Harish Patil, Aamer Jaleel, Greg Hamerly, and Brad Calder , **Cross Binary Simulation Points** , International Symposium on Performance Analysis of Systems and Software (ISPASS), April 2007 (pdf)

- Satish Narayanasamy, Ayse Coskun, and Brad Calder , **Transient Fault Prediction Based on Anomalies in Processor Events** , Design Automation and Test in Europe (DATE), April 2007 (pdf)

- Weifeng Zhang, Brad Calder and Dean Tullsen , **Accelerating and Adapting Precomputation Threads for Efficient Prefetching** , In the International Symposium on High Performance Computer Architecture, February 2007. (pdf)

- Wei Chuang, Satish Narayanasamy, and Brad Calder , **Bounds Checking with Taint-Based Analysis** , International Conference on High Performance Embedded Architectures & Compilers, January 2007 (pdf)

2006

- Jack Sampson, Ruben Gonzalez, Jean-Francois Collard, Norm Jouppi, Mike Schlansker and Brad Calder , **Exploiting Fine-Grained Data Parallelism with Chip Multiprocessors and Fast Barriers** , In proceedings of the International Symposium on Microarchitecture (MICRO), December 2006. (pdf)

- Wei Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Osvaldo Colavin, and Brad Calder , **Unbounded Page-Based Transactional Memory** , International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Oct 2006 (pdf)

- Satish Narayanasamy, Cristiano Pereira and Brad Calder , **Recording Shared Memory Dependencies Using Strata** , International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 2006 (pdf)

- Satish Narayanasamy, Bruce Carneal and Brad Calder , **Patching Processor Design Errors** , International Conference on Computer Design, October 2006 (pdf)

- Weifeng Zhang, Steve Checkoway, Brad Calder, and Dean Tullsen , **Speculative Code Value Specialization Using the Trace Cache Fill Unit** , International Conference on Computer Design, Oct 2006 (pdf)

- Satish Narayanasamy, Cristiano Pereira and Brad Calder, , **Software Profiling for Deterministic Replay Debugging of User Code** , The 5th International Conference on Software Methodologies, Tools and Techniques (SoMeT), October 2006 (pdf)

- Michael Van Biesbrouck, Lieven Eeckhout, and Brad Calder, **Efficient Sampling Startup for SimPoint** , IEEE Micro Special Issue Jul/Aug 06 Modeling & Simulation (pdf)

- Jeremy Lau, Matt Arnold, Micheal Hind, and Brad Calder , **Online Performance Auditing: Using Hot Optimizations Without Getting Burned** , ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2006 (pdf)

- Satish Narayanasamy, Cristiano Pereira, Harish Patil, Robert Cohn, and Brad Calder , **Automatic Logging of Operating System Effects to Guide Application Level Architecture Simulation** , ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems (Sigmetrics), June 2006 (pdf)

- Greg Hamerly, Erez Perelman, Jeremy Lau, Brad Calder, and Timothy Sherwood , **Using Machine Learning to Guide Architecture Simulation** , Journal of Machine Learning Research (JMLR), 2006 (pdf)

- Erez Perelman, Marzia Polito, Jean-Yves Bouguet, John Sampson, Brad Calder, Carole Dulong , **Detecting Phases in Parallel Applications on Shared Memory Architectures** , IEEE International Parallel and Distributed Processing Symposium (IPDPS), April 2006. (pdf)

- Greg Hamerly, Erez Perelman, and Brad Calder, **Comparing Multinomial and K-Means Clustering for SimPoint** , International Symposium on Performance Analysis of Systems and Software (ISPASS), March 2006. (pdf)

- Michael Van Biesbrouck, Lieven Eeckhout, and Brad Calder, **Considering All Starting Points**

for Simultaneous Multithreading Simulation , International Symposium on Performance Analysis of Systems and Software (ISPASS), March 2006. (pdf)

- Jeremy Lau, Erez Perelman, and Brad Calder, **Selecting Software Phase Markers with Code Structure Analysis** , International Symposium on Code Generation and Optimization (CGO), March 2006. (pdf)

- Weifeng Zhang, Brad Calder and Dean Tullsen, **A Self Repairing Prefetcher in an Event-Driven Dynamic Optimization Framework** , International Symposium on Code Generation and Optimization (CGO), March 2006. (pdf)

2005

- Satish Narayanasamy, Gilles Pokam, and Brad Calder **BugNet: Recording Application Level Execution for Deterministic Replay Debugging** , IEEE Micro: Micro's Top Picks from Computer Architecture Conferences, December 2005 (pdf)

- Michael Van Biesbrouck, Lieven Eeckhout, and Brad Calder, **Efficient Sampling Startup for Sampled Processor Simulation** , 2005 International Conference on High Performance Embedded Architectures & Compilers, November 2005. (pdf)

- Bengu Li, Ganesh Venkatesh, Brad Calder, and Rajiv Gupta, **Exploiting a Computation Reuse Cache to Reduce Energy in Network Processors** , 2005 International Conference on High Performance Embedded Architectures & Compilers, November 2005. (pdf)

- Lieven Eeckhout, John Sampson, and Brad Calder, **Exploiting Program Microarchitecture Independent Characteristics and Phase Behavior for Reduced Benchmark Suite Simulation** , 2005 IEEE International Symposium on Workload Characterization, October 2005 (pdf)

- Cristiano Pereira, Jeremy Lau, Brad Calder, and Rajesh Gupta, **Dynamic Phase Analysis for Cycle-Close Trace Generation** , International Conference on Hardware/Software Codesign and System Synthesis, September 2005 (pdf)

- Weifeng Zhang, Brad Calder, and Dean Tullsen, **An Event-Driven Multithreaded Dynamic Optimization Framework** , International Conference on Parallel Architectures and Compilation Techniques, September 2005. (pdf)

- Erez Perelman, Trishul Chilimbi, and Brad Calder, **Variational Path Profiling** , International Conference on Parallel Architectures and Compilation Techniques, September 2005. (pdf)

- Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder, **SimPoint 3.0: Faster and More Flexible Program Analysis** , Journal of Instruction Level Parallelism, September 2005. (pdf)

- Brad Calder, Andrew Chien, Ju Wang and Don Yang, **The Entropia Virtual Machine for Desktop Grids** , International Conference on Virtual Execution Environments, June 2005. (pdf)

- Satish Narayanasamy, Gilles Pokam, and Brad Calder, **BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging** , International Symposium on Computer Architecture, June 2005. (pdf)

- Satish Narayanasamy, Hong Wang, Perry Wang, John Shen, Brad Calder, **A Dependency Chain Clustered Microarchitecture** , IEEE International Parallel and Distributed Processing Symposium, April 2005. (pdf)

- Jeremy Lau, Jack Sampson, Erez Perelman, Greg Hamerly, and Brad Calder, **The Strong Correlation between Code Signatures and Performance** , IEEE International Symposium on Performance Analysis of Systems and Software, March 2005. (pdf)

- Jeremy Lau, Erez Perelman, Greg Hamerly, Timothy Sherwood, and Brad Calder, **Motivation for Variable Length Intervals and Hierarchical Phase Behavior** , IEEE International Symposium on Performance Analysis of Systems and Software, March 2005. (pdf)

- Jeremy Lau, Stefan Schoenmackers, and Brad Calder, **Transition Phase Classification and Prediction** , In the 11th International Symposium on High Performance Computer Architecture, February 2005. (pdf)

2004

- Nathan Tuck, Brad Calder and George Varghese, , **Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow** , 37th International Symposium on Microarchitecture, December 2004. (pdf)

- Eric Tune, Rakesh Kumar, Dean Tullsen and Brad Calder , **Balanced Multithreading: Increasing Throughput via a Low Cost Multithreading Hierarchy** , 37th International Symposium on Microarchitecture, December 2004. (pdf)

- Timothy Sherwood, Mark Oskin, and Brad Calder , **Balancing Design Options with Sherpa** , International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), September 2004, (pdf)

- Brad Calder, Todd Austin, Don Yang, Timothy Sherwood, Suleyman Sair, David Newquist and Tim Cusac , **BitRaker Anvil: Binary Instrumentation for Rapid Creation of Simulation and Workload Analysis Tools** , Proceedings of Global Signal Processing (GSPx), September, 2004, (pdf)

- Greg Hamerly, Erez Perelman, and Brad Calder, **How to Use SimPoint to Pick Simulation Points** ACM SIGMETRICS Performance Evaluation Review, Volume 31(4), March 2004 (pdf)

- Glenn Reiman and Brad Calder, **Using a Serial Cache for Energy Efficient Instruction Fetching** , Journal of Systems Architecture, 2004, (pdf)

- Jeremy Lau, Stefan Schoenmackers, and Brad Calder, **Structures for Phase Classification,** 2004 IEEE International Symposium on Performance Analysis of Systems and Software, March 2004 (pdf)

- Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder, **A Co-Phase Matrix to Guide Simultaneous Multithreading Simulation,** 2004 IEEE International Symposium on Performance Analysis of Systems and Software, March 2004 (pdf)

- Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese, **Deterministic Memory-**

Efficient String Matching Algorithms for Intrusion Detection , Proceedings of the IEEE Infocom Conference, Hong Kong, China, March 2004. (pdf)

- Satish Narayanasamy, Yuanfang Hu, Suleyman Sair, and Brad Calder, **Creating Converged Trace Schedules Using String Matching** , In the 10th International Symposium on High Performance Computer Architecture, February 2004. (pdf)

2003

- Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder, **Discovering and Exploiting Program Phases,** IEEE Micro: Micro's Top Picks from Computer Architecture Conferences, December 2003 (pdf)

- Jeremy Lau, Stefan Schoenmackers, Timothy Sherwood, and Brad Calder, **Reducing Code Size With Echo Instructions** , International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, October 2003. (pdf)

- Erez Perelman, Greg Hamerly, and Brad Calder, **Picking Statistically Valid and Early Simulation Points** , International Conference on Parallel Architectures and Compilation Techniques, September 2003. (pdf)

- Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder, **Using SimPoint for Accurate and Efficient Simulation,** , ACM SIGMETRICS the International Conference on Measurement and Modeling of Computer Systems, June 2003. (pdf)

- Timothy Sherwood, George Varghese, and Brad Calder, **A Pipelined Memory Architecture for High Throughput Network Processors** , 30th International Symposium on Computer Architecture, June 2003. (pdf)

- Timothy Sherwood, Suleyman Sair, and Brad Calder, **Phase Tracking and Prediction,** 30th International Symposium on Computer Architecture, June 2003. (pdf)

- Weihaw Chuang and Brad Calder, **Predicate Prediction for Efficient Out-of-order Execution,** 16th Annual ACM International Conference on Supercomputing, June 2003. (pdf)

- Weihaw Chuang, Brad Calder, and Jeanne Ferrante, **Phi Predication for Light Weight If-Conversion** , International Symposium on Code Generation and Optimization, March 2003. (pdf)

- Suleyman Sair, Timothey Sherwood and Brad Calder, **A Decoupled Predictor-Directed Stream Prefetching Architecture** , In the IEEE Transactions on Computers, Vol. 52, No. 5, March 2003 (pdf)

- Andrew Chien, Brad Calder, Stephen Elbert, and Karan Bhatia , **Entropia: Architecture and Performance of an Enterprise Desktop Grid System** , Journal of Parallel Distributed Computing, Vol 63, Issue 5, May 2003, pages 597-610. (pdf)

- Satish Narayanasamy, Timothy Sherwood, Suleyman Sair, Brad Calder, George Varghese, **Catching Accurate Profiles in Hardware** , In the 9th International Symposium on High Performance Computer Architecture, February 2003. (pdf)

- Beth Simon, Brad Calder, and Jeanne Ferrante, **Incorporating Predicate Information Into Branch Predictors** , 9th International Symposium on High Performance Computer Architecture, February 2003. (pdf)

2002

- Jamison Collins, Suleyman Sair, Brad Calder, and Dean Tullsen, **Pointer-Cache Assisted Prefetching** , To appear in the 35th Annual International Symposium on Microarchitecture, November 2002. (pdf)

- Timothy Sherwood, Erez Perelman, Greg Hamerly and Brad Calder, **Automatically Characterizing Large Scale Program Behavior** , In the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, October 2002. (pdf)

- Eric Tune, Dean Tullsen, and Brad Calder, **Quantifying Instruction Criticality,** in the Eleventh International Conference on Parallel Architectures and Compilation Techniques, September 2002. (pdf)

- Lori Carter and Brad Calder, **Using Predicate Path Information in Hardware to Determine True Dependences** , In the proceedings of the 16th Annual ACM International Conference on Supercomputing, June 2002. (pdf)

- Lori Carter, Weihaw Chuang, and Brad Calder **An EPIC Processor with Pending Functional Units** , In the proceedings of the 4th International Symposium on High Performance Computing (ISHPC2K), May 2002, (c) Springer-Verlag . (pdf)

- Suleyman Sair, Timothy Sherwood and Brad Calder, **Quantifying Load Stream Behavior,** , 8th International Symposium on High-Performance Computer Architecture, February 2002. (pdf)

2001

- Timothy Sherwood and Brad Calder, **Patchable Instruction ROM Architecture,** , International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), November 2001. (pdf)

- Timothy Sherwood, Erez Perelman and Brad Calder, **Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications** , International Conference on Parallel Architectures and Compilation Techniques, September 2001. (pdf)

- Chandra Krintz and Brad Calder, **Reducing Delay with Dynamic Selection of Compression Formats** Tenth International Symposium on High Performance Distributed Computing, August 2001. (pdf)

- Timothy Sherwood and Brad Calder, **Automated Design of Finite State Machine Predictors for Customized Processors** , 28th International Symposium on Computer Architecture, June 2001. (pdf)

- Chandra Krintz and Brad Calder, **Using Annotations to Reduce Dynamic Optimization Time,** ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI), June 2001. (pdf)

- Glenn Reinman, Brad Calder, and Todd Austin, **Optimizations Enabled by a Decoupled Front-End Architecture**, IEEE Transactions on Computers, Vol. 50, No. 4, April 2001. (pdf)

- Chandra Krintz, David Grove, Vivek Sarkar, and Brad Calder, **Reducing the Overhead of Dynamic Compilation**, Software: Practice and Experience, pages 717-738, Volume 31, Issue 8, March 2001. (pdf)

- Eric Tune, Dongning Liang, Dean M. Tullsen, and Brad Calder, **Dynamic Prediction of the Critical Dependence Path** , 7th International Symposium On High Performance Computer Architecture, January 2001. (pdf)

2000

- Timothy Sherwood, Suleyman Sair, and Brad Calder, **Predictor-Directed Stream Buffers**, In proceedings of the 33rd International Symposium on Microarchitecture, December 2000. (pdf)

- Lori Carter, Beth Simon, Brad Calder, Larry Carter, and Jeanne Ferrante, **Path Analysis and Renaming for Predicated Instruction Scheduling** International Journal of Parallel Programming, pages 563-588, December 2000. (pdf)

- Timothy Sherwood and Brad Calder, **Loop Termination Prediction**, In the proceedings of the 3rd International Symposium on High Performance Computing (ISHPC2K), October 2000, (c) Springer-Verlag . (pdf)

- Barbara Kreaseck, Dean Tullsen, and Brad Calder, **Limits of Task-based Parallelism in Irregular Applications**, In the proceedings of the 3rd International Symposium on High Performance Computing (ISHPC2K), October 2000, (c) Springer-Verlag . (pdf)

- Timothy Sherwood and Brad Calder, **ToolBlocks: An Infrastructure for the Construction of Memory Hierarchy Analysis Tools** . In the proceedings of the International European Conference on Parallel Computing (EURO-PAR), August 2000. (pdf)

- Timothy Sherwood and Brad Calder, **. Automated Design of Finite State Machine Predictors for Value Prediction and Branch Prediction** , UCSD Techreport, CS2000-0656, June 2000 (pdf)

- Brad Calder and Glenn Reinman, **A Comparative Survey of Load Speculation Architectures** , Journal of Instruction Level Parallelism, May 2000. (pdf)

1999

- Glenn Reinman, Brad Calder, and Todd Austin, **Fetch Directed Instruction Prefetching** . In proceedings of the 32nd International Symposium on Microarchitecture, November 1999. (pdf)

- Chandra Krintz, Brad Calder, and Urs Hölzle, **Reducing Transfer Delay Using Java Class File Splitting and Prefetching** . In proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, November 1999. (pdf)

- Lori Carter, Beth Simon, Brad Calder, Larry Carter, and Jeanne Ferrante, **Predicated Static Single Assignment** . In proceedings of the International Conference on Parallel Architectures and

Compilation Techniques, October 1999. (pdf)

- Timothy Sherwood and Brad Calder, **Time Varying Behavior of Programs**, UC San Diego Technical Report UCSD-CS99-630, August 1999. (pdf)

- Timothy Sherwood, Brad Calder, and Joel Emer, **Reducing Cache Misses Using Hardware and Software Page Placement** , In the ACM International Conference on Supercomputing, pages 155-164, June 1999. (pdf)

- Glenn Reinman, Brad Calder, Dean Tullsen, Gary Tyson, and Todd Austin, **Classifying Load and Store Instructions for Memory Renaming,** In the ACM International Conference on Supercomputing, pages 399-407, June 1999. (pdf)

- Glenn Reinman, Todd Austin, and Brad Calder, **A Scalable Front-End Architecture for Fast Instruction Delivery** , 26th International Symposium on Computer Architecture, pages 234-245, May 1999. (pdf)

- Brad Calder, Glenn Reinman, and Dean Tullsen, **Selective Value Prediction** , 26th International Symposium on Computer Architecture, pages 64-73, May 1999. (pdf)

- Brad Calder, Peter Feller, and Alan Eustace, **Value Profiling and Optimization**, Journal of Instruction Level Parallelism, March 1999. (pdf)

- Steven Wallace, Dean Tullsen, and Brad Calder, **Instruction Recycling on a Multiple-Path Processor** , 5th International Symposium On High Performance Computer Architecture, pages 44-53, January 1999. (pdf)

- Brad Calder and Dirk Grunwald, **The Precomputed Branch Architecture** , Journal of Systems Architecture, pages 651-679, Vol. 45, 1999. (pdf)

1998

- Glenn Reinman and Brad Calder, **Predictive Techniques for Aggressive Load Speculation** , 31st International Symposium on Microarchitecture, pages 127-137, December 1998. (pdf)

- Brad Calder, Chandra Krintz, Simmi John, and Todd Austin, **Cache-Conscious Data Placement** , 8th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, pages 139-149, October 1998. (pdf)

- Chandra Krintz, Brad Calder, Han Bok Lee, and Benjamin G. Zorn, **Overlapping Execution with Transfer Using Non-Strict Execution for Mobile Programs** , 8th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, pages 159-169, October 1998. (pdf)

- Artur Klauser, Todd Austin, Dirk Grunwald, and Brad Calder, **Dynamic Hammock Predication for Non-predicated Instruction Set Architectures**, International Conference on Parallel Architectures and Compilation Techniques, Paris, France, pages 278-285, October 1998. (pdf)

- Dean Tullsen and Brad Calder, **Computing Along the Critical Path**, UC San Diego Technical Report, October 1998. (pdf)

- Iris Bahar, Brad Calder and Dirk Grunwald, **A Comparison of Software Code Reordering and Victim Buffers**, Third Workshop on Interaction Between Compilers and Computer Architectures, October 1998. (pdf)

- Steven Wallace, Brad Calder, and Dean Tullsen, **Threaded Multiple Path Execution** , 25th International Symposium on Computer Architecture, pages 238-249, June 1998. (pdf)

1997

- Brad Calder, Peter Feller, and Alan Eustace, **Value Profiling**, 30th International Symposium on Microarchitecture, pages 259-269, December 1997. (pdf)

- Nikolas Gloy, Trevor Blackwell, Michael D. Smith, and Brad Calder, **Procedure Placement Using Temporal Ordering Information**, 30th International Symposium on Microarchitecture, pages 303-313, December 1997. (pdf)

- Amir H. Hashemi, David R. Kaeli, and Brad Calder, **Efficient Procedure Mapping Using Cache Line Coloring**, Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, pages 171-182, June 1997. (pdf)

- Amir H. Hashemi, David R. Kaeli, and Brad Calder, **Procedure Mapping Using Static Call Graph Estimation**, Workshop on the Interaction between Compilers and Computer Architectures, San Antonio, Texas, February 1997 (pdf)

- Brad Calder, Dirk Grunwald, Donald Lindsay, Michael Jones, James Martin, Michael Mozer, and Benjamin Zorn, **Evidence-based Static Branch Prediction using Machine Learning** , ACM Transactions on Programming Languages and Systems, pages 188-222, Vol. 19, No. 1, January 1997. (pdf)

1996

- Brad Calder, Dirk Grunwald, and Joel Emer, **Predictive Sequential Associative Cache**, 2nd International Symposium on High Performance Computer Architecture, pages 244-253, February 1996. (pdf)

1995

- Brad Calder, Dirk Grunwald, and Amitabh Srivastava, **The Predictability of Branches in Libraries** , 28th International Symposium on Microarchitecture, pages 24-34, November 1995, WRL Research Report 95/6 version. (pdf)

- Brad Calder, Dirk Grunwald, and Joel Emer, **A System Level Perspective on Branch Architecture Performance** , 28th International Symposium on Microarchitecture, pages 199-206, November 1995. (pdf)

- Brad Calder and Dirk Grunwald, **Next Cache Line and Set Prediction** , 22nd International Symposium on Computer Architecture, pages 287-296, June 1995. (pdf)

- Dennis Lee, Jean-Loup Baer, Brad Calder, and Dirk Grunwald, **Instruction Cache Fetch Policies for Speculative Execution** , 22nd International Symposium on Computer Architecture, pages

357-367, June 1995. (pdf)

- Brad Calder, Dirk Grunwald, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn, **Corpus-based Static Branch Prediction** , Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, pages 79-92, June 1995. (pdf)

1994

- Brad Calder, Dirk Grunwald and Benjamin Zorn, **Quantifying Behavioral Differences Between C and C++ Programs** , Journal of Programming Languages, pages 313-351, Vol 2, Num 4, 1994. (pdf)
- Brad Calder and Dirk Grunwald, **Reducing Branch Costs via Branch Alignment** , 6th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 242-251, October 1994. (pdf)
- Brad Calder and Dirk Grunwald, **Fast and Accurate Instruction Fetch and Branch Prediction** , 21st International Symposium on Computer Architecture, pages 2-11, April 1994. (pdf)
- Brad Calder and Dirk Grunwald, **Reducing Indirect Function Call Overhead In C++ Programs** , 21st Symposium on Principles of Programming Languages, pages 397-408, January 1994. (pdf)

1993

- Dave Wagner and Brad Calder, **Leapfrogging: A Portable Technique for Implementing Efficient Futures** , 4th ACM Principles and Practice of Parallel Processing, pages 208-217, May 1993. (pdf)

Thesis (Back To Top)

Peter Feller M.S., Value Profiling for Instructions and Memory Locations , April 1998

Chandra Krintz Ph.D., Reducing Load Delay to Improve Performance of Internet-Computing Programs, May 2001

Glenn Reinman Ph.D., Hardware Optimizations Enabled by a Decoupled Fetch Architecture, June 2001

Beth Simon Ph.D., Turning Predicate Information to Advantage to Improve Compiler Scheduling and Branch Prediction , December 2001.

Lori Carter , Ph.D., Compiler and Hardware Predicated Dependency Analysis and Scheduling, February 2002.

Timothy Sherwood , Ph.D., Application-Tuned Processor Architectures , June 2003.

Suleyman Sair , Ph.D., Predictor-Directed Data Prefetching for Pointer-based Applications, June 2003.

Brad Calder, **Hardware and Software Mechanisms for Instruction Fetch Prediction,** Dissertation, University of Colorado Technical Report CU-CS-781-95, August 1995.

EXHIBIT 4
TO DECLARATION OF
JAMES A FLIGHT

In Proceedings of 30th International Symposium on Computer Architecture (ISCA), June 2003.

Phase Tracking and Prediction

Timothy Sherwood Suleyman Sair Brad Calder

Department of Computer Science and Engineering
University of California, San Diego
{sherwood,ssair,calder}@cs.ucsd.edu

Abstract

In a single second a modern processor can execute billions of instructions. Obtaining a bird's eye view of the behavior of a program at these speeds can be a difficult task when all that is available is cycle by cycle examination. In many programs, behavior is anything but steady state, and understanding the patterns of behavior, at run-time, can unlock a multitude of optimization opportunities.

In this paper, we present a unified profiling architecture that can efficiently capture, classify, and predict phase-based program behavior on the largest of time scales. By examining the proportion of instructions that were executed from different sections of code, we can find generic phases that correspond to changes in behavior across many metrics. By classifying phases generically, we avoid the need to identify phases for each optimization, and enable a unified prediction scheme that can forecast future behavior. Our analysis shows that our design can capture phases that account for over 80% of execution using less that 500 bytes of on-chip memory.

1 Introduction

Modern processors can execute upwards of 5 billion instructions in a single second, yet most architectural features target program behavior on a time scale of hundreds to thousands of instructions, less than half a μS. While these optimizations can provide large benefits, they are limited in their ability to see the program behavior in a larger context.

Recently there has been a renewed interest in examining the run-time behavior of programs over longer periods of time [10, 11, 19, 20, 3]. It has been shown that programs can have considerably different behavior depending on which portion of execution is examined. More specifically, it has been shown that many programs execute as a series of phases, where each phase may be very different from the others, while still having a fairly homogeneous behavior within a phase. Taking advantage of this time varying behavior can lead to, among other things, improved power management, cache control, and more efficient simulation. The primary goal of this research is the development of a unified run-time phase detection and prediction mechanism that can be used to guide any optimization seeking to exploit large scale program behavior.

A phase of program behavior can be defined in several ways. Past definitions are built around the idea of a phase being an interval of execution during which a measured program metric is relatively stable. We extend this notion of a phase to include all similar sections of execution regardless of temporal adjacency. Simply put, if a phase of execution is correctly identified, there should only be small variations between any two execution intervals identified as being part of the same phase. A key point of this paper is that the phase behavior seen in any program metric is directly a function of the way the code is being executed. If we can accurately capture this behavior at run-time through the computation of a *single* metric, we can use this to guide many optimization and policy decisions without duplicating phase detection mechanisms for *each* optimization.

In this paper, we present an efficient run-time phase tracking architecture that is based on detecting changes in the *proportions* of the code being executed. In addition, we present a novel phase prediction architecture that can predict, not only when a phase change is about to occur, but also the phase to which it is will transition. Since our phase tracking implementation is based upon code execution frequencies, it is independent of any individual architecture metric. This allows our phase tracker to be used as a general profiling technique building up a profile or database of architecture information on a per phase basis to be used later for hardware or software optimization. Independence from architecture metrics allows us to consistently track phase information as the program's behavior changes due to phase-based optimizations.

We demonstrate the effectiveness of our hardware based phase detection and classification architecture at automatically partitioning the behavior of the program into homogeneous phases of execution and to identify phase changes. We show that the changes in many important metrics, such as IPC and energy, correlate very closely with the phase changes found by our metric. We then evaluate the effectiveness of phase tracking and prediction for value profiling, data cache reconfiguration, and re-configuring the width of the processor.

The rest of the paper is laid out as follows. In Section 2, prior work related to phase-based program behavior is discussed. Simulation methodology and benchmark descriptions can be found in Section 3. Section 4 describes our phase tracking architecture. The design and evaluation of the phase predictor are found in Section 5. Section 6 presents several potential applications of our phase tracking architecture. Finally, the results are summarized in Section 7.

2 Related Work

In this Section we describe work related to phase identification and phase-based optimization.

In [19], we provided an initial study into the time varying behavior of programs, showing that programs have repeatable phase-based behavior over many hardware metrics – cache behavior, branch prediction, value prediction, address prediction,

IPC and RUU occupancy for all the SPEC 95 programs. Looking at these metrics over time, we found that many programs have repeating patterns, and that important metrics tend to change at the same time. These places represent phase boundaries.

In [20], we proposed that by profiling only the code that was executed over time we could automatically identify periodic and phase behavior in programs. The goal was to automatically find the repeating patterns observed in [19], and the lengths (periods) of these patterns. We then extended this work in [21], using techniques from machine learning to break the complete execution of the program into phases (clusters) by only tracking the code executed. We found that intervals of execution grouped into the same phase had similar behavior across all the architecture metrics examined. From this analysis, we created a tool called SimPoint [21], which automatically identifies a small set of intervals of execution (simulation points) in a program to perform architecture simulations. These simulation points provide an accurate and efficient representation of the complete execution of the program.

The work of Dhodapkar and Smith [10, 9] is the most closely related to ours. They found a relationship between phases and instruction working sets, and that phase changes occur when the working set changes. They propose that by detecting phases and phase changes, multi-configuration units can be re-configured in response to these phase changes. They have used their working set analysis for instruction cache, data cache and branch predictor re-configuration to save energy [10, 9].

The work we present in this paper identifies phases and phase changes by keeping track of the proportions in which the code was executed during an interval based upon the profiler used in [20]. In comparison, Dhodapkar and Smith [10, 9] track the phase and phase changes solely upon what code was executed (working set), without weighting the code by its frequency of execution. Future research is needed to compare these two approaches.

Additional differences between our work include our examination of architectures for predicting phase changes, and different uses from [10, 9], such as value profiling and processor width reconfiguration. We provide an architecture that can fairly accurately predict what the next phase will be, along with predicting when there will be a phase change. In comparison, Dhodapkar and Smith do not examine phase-based prediction [10, 9], but concentrate on detecting when the working set size changes, and then reactively apply optimization.

Merten et al. [15] developed a run-time system for dynamically optimizing frequently executed code. Then in [3], Barnes et al. extend this idea to perform phase-directed complier optimizations. The main idea is the creation of optimized code "packages" that are targeted towards a given phase, with the goal of execution staying within the package for that phase. Barnes et al. concentrate primarily on the compiler techniques needed to make phase-directed compiler optimizations a reality, and do not examine the mechanics of hardware phase detection and classification. We believe that using the techniques in [3] in conjunction with our phase classification and prediction architecture will provide a powerful run-time execution environment.

I Cache	16k 4-way set-associative, 32 byte blocks, 1 cycle latency
D Cache	16k 4-way set-associative, 32 byte blocks, 1 cycle latency
L2 Cache	128K 8-way set-associative, 64 byte blocks, 12 cycle latency
Main Memory	120 cycle latency
Branch Pred	hybrid - 8-bit gshare w/ 2k 2-bit predictors + a 8k bimodal predictor
O-O-O Issue	out-of-order issue of up to 4 operations per cycle, 64 entry re-order buffer
Mem Disambig	load/store queue, loads may execute when all prior store addresses are known
Registers	32 integer, 32 floating point
Func Units	2-integer ALU, 2-load/store units, 1-FP adder, 1-integer MULT/DIV, 1-FP MULT/DIV
Virtual Mem	8K byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete

Table 1: Baseline Simulation Model.

3 Methodology

To perform our study, we collected information for ten SPEC 2000 programs `applu`, `apsi`, `art`, `bzip`, `facerec`, `galgel`, `gcc`, `gzip`, `mcf`, and `vpr` all with reference inputs. All programs were executed from start to completion using SimpleScalar [5] and Wattch [4]. Because of the lengthy simulation time incurred by executing all of the programs to completion, we chose to focus on only 10 programs. We chose the above 10 programs since their phase based behavior represents a reasonable snapshot of the SPEC 2000 benchmark suite, along with picking some of the programs that showed the most interesting phase-based behavior. Each program was compiled on a DEC Alpha AXP-21164 processor using the DEC C, and FORTRAN compilers. The programs were built under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`).

The timing simulator used was derived from the SimpleScalar 3.0 tool set [5], a suite of functional and timing simulation tools for the Alpha AXP ISA. The baseline microarchitecture model is detailed in Table 1. In addition to this, we wanted to examine energy usage optimizations, so we used a version of Wattch [4] to capture this information. We modified all of these tools to log and reset the statistics every 10 million instructions, and we use this as a base for evaluation.

4 Phase Capture

In this section we motivate the occurrence of phase-based behavior, describe our architecture for capturing it, and examine the accuracy of using the program behavior in our phase-tracking architecture to identify phase changes for various hardware metrics.

4.1 Phase-Based Behavior

The goal of this research is to design an efficient and general purpose technique for capturing and predicting the run-time phase behavior of programs for the purpose of guiding any optimization seeking to exploit large scale program behavior. Figure 1 helps to motivate our approach to the problem. This figure shows the behavior of two programs, `gcc` and `gzip`, as measured by various different statistics over the course of their execution from start to finish. Each point on the graph is taken over 10 million instructions worth of execution. The metrics shown are the

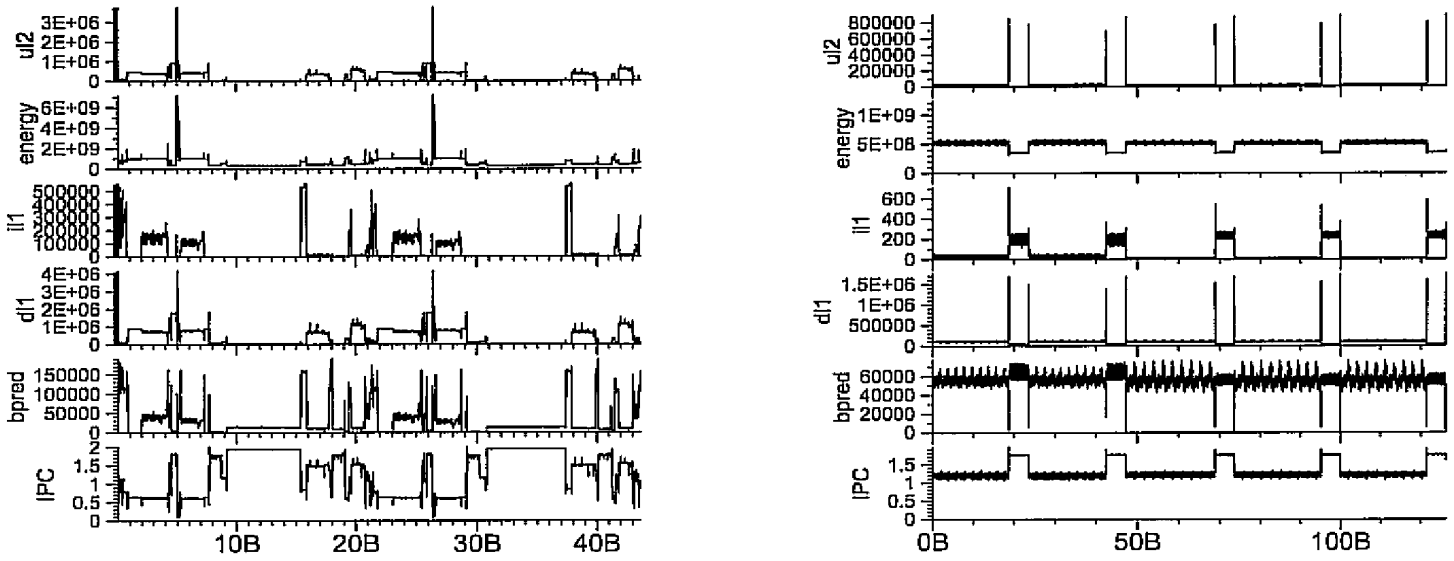

Figure 1: To illustrate the point that phase changes happen across many metrics all at the same time, we have plotted the value of these metrics over billions of instructions executed for the programs `gcc` *(shown left) and* `gzip` *(shown right). Each point on the graph is an average over 10 million instructions. The number of unified L2 cache misses (ul2), the energy consumed by the execution of the instructions, the number of instruction cache (il1) misses, the number of data cache misses (dl1), the number of branch mispredictions (bpred) and the average IPC are plotted.*

number of unified L2 cache misses (ul2), the energy consumed by the execution of the instructions, the number of instruction cache (il1) misses, the number of data cache misses (dl1), the number of branch mispredictions (bpred) and the average IPC. The results show that all of the metrics tend to change in unison, although not necessarily in the same direction. In addition to this, patterns of recurring behavior can be seen over very large time scales.

As can be seen from these graphs, even at a granularity of 10 million instructions (which is at the same time scale as operating system time slices) there can be wildly different behavior seen between intervals. In this paper, we concentrate on a granularity of 10 million instructions because it is both outside the scope of normal architectural timing and is small enough to allow for many complex phase behaviors to be seen.

4.2 Tracking Phases by Executed Code

Our phase tracker architecture operates at two different time scales. It gathers profile information very quickly in order to keep up with processor speeds, while at the same time it compares any data it gathers with information collected over the long term. Additionally, it must be able to do all that while still being reasonable in size.

Our phase profile generation architecture can be seen in Figure 2. The key idea is to capture basic block information during execution, while not relying on any compiler support. Larger basic blocks need to be weighed more heavily as they account for a more significant portion of the execution. To *approximate* gathering basic block information, we capture branch PCs and the number of instructions executed between branches. The input to the architecture is a tuple of information: a branch identifier (PC) and the number of instructions since the last branch PC was executed. This allows us to roughly capture each basic block executed along with the weight of the basic block in terms of the number of instructions executed, as we did in [20, 21] for identifying simulation points.

Classifying phases by examining only the code that is executed allows our phase tracker to be independent of any individual architecture metric. This allows our phase tracker to be used as a general profiling technique building up a profile or database of architecture information on a per phase basis to be used later for hardware or software optimization. Independence from architecture metrics is also very important to allow us to consistently track phase information as the program's behavior changes due to phase-based optimizations.

At this point it is worth making more explicit the differences between our technique and that of Dhodapkar and Smith [10, 9]. Dhodapkar and Smith use a bit vector to track the working set of the code for a particular interval. While our technique is based on the basic block vectors used in [20]. The bit vectors of Dhodapkar and Smith track a metric that is related to which code blocks were *touched*, whereas our metric tracks the *proportion* of time spent executing in each code block. This is a subtle but important distinction. We have found that in complex programs (such as `gcc` and `gzip`) there are many instructions blocks that execute only intermittently. When tracking the pure working set, these infrequently executed blocks can disguise the frequently executed blocks that dominate the behavior of the application. On the other hand, by tracking the frequency of code execution it is possible to distinguish important instructions (basic blocks) from a sea of infrequently executed ones. Examining these differences in more detail is a topic of future research.

Another advantage of tracking the proportions in which the basic blocks are executed is that we can use this information to

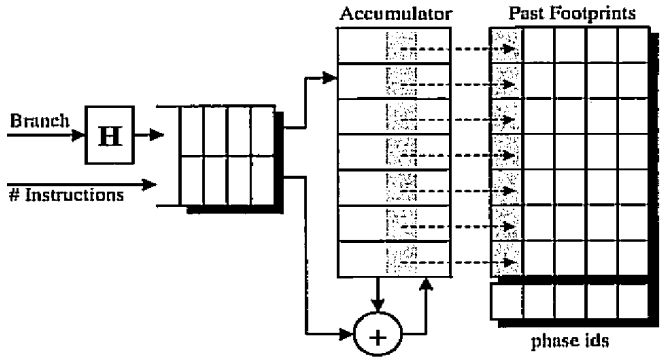

Figure 2: Our phase classification architecture. Each branch PC is captured along with the number of instructions from the last branch. The bucket entry corresponding to a hash of the branch PC is incremented by the number of instructions. After each profiling interval has completed, this information is classified, and if it is found to be unique enough, stored in the past footprint table along with its phase ID.

identify not only when different sections of code are executing, but also when those sections of code are being exercised differently. A simple example is in a graphics manipulation program running a parameterized filter on an input image. If you run a simple 3x3 blur filter on an image you get very different behavior than if you run a 7x7 filter on the same image despite the fact that the same filter code is executing. The 7x7 filter will have many more memory references and those memory references conflict very differently in the cache than in the 3x3 case. We have seen this very behavior in examining the interactive graphics program `xv`. Using the proportion of execution for each basic block can distinguish these differences, because in the 3x3 filter the head of the loop is called more than twice as frequently as in the 7x7 filter.

The same general idea applies to other data structures as well. Take for example a linked list. As the number of nodes in the linked list traversal changes over different loop invocations, the number of instructions executed inside the loop versus the time spent outside the loop also changes. This behavior can be captured when including a measure of the proportion of the code executed, and this can distinguish between link list traversals of different lengths.

4.3 Capturing the Code Profile

To index into the accumulator table in Figure 2, the branch PC is reduced to a number from 1 to $Nbuckets$ using a hash function. We have found that 32 buckets is sufficient to distinguish between different phases even for some of the more complex programs such as `gcc`. A counter is kept for each bucket, and the counter is incremented by the number of instructions from the last branch to the current branch being processed. Each accumulator table entry is a large (in this study 24-bit), saturating counter, which will not saturate during our profiling interval of 10 million instructions. Updating the accumulator table is the only operation that needs to be performed at a rate equivalent to the processor's execution of the program (once for every branch executed). In comparison, the phase classification described below needs to only be performed once every 10 million instructions (at the end of each interval), and thus is not nearly as performance critical.

We note that the hashing function we use is fundamentally the same as the random projection method we used to generate phases in [21]. In this prior work, we make use of random projections of the data to reduce the dimensionality of the samples being taken. A random projection takes trace data in the form of a matrix of size $L \times B$, where L is the length of the trace and B is the number of unique basic blocks, and multiplies it by a random matrix of size $B \times N$, where N is the desired dimensionality of the data which is much smaller than B. This creates a new matrix of size $L \times N$, which has clustering properties very similar to the original data. The random projection method is a powerful technique when used with clustering algorithms, and for capturing phase behavior as we showed in [21]. The hashing scheme we use in this paper is essentially a degenerate form of random projection that makes a hardware implementation feasible while still having low error. If all the elements of the random projection matrix consist of either a 0 or a 1, and they are placed such that no column of the matrix contains more than a single 1, then the random projection is identical to this simple hashing mechanism. We have designed our phase classification architecture around this principle.

Figure 3 shows the effect of applying the above mentioned technique for capturing the phase behavior of the integer benchmark `gzip`. The x-axis of the figure is in billions of instructions, as is the case in Figure 1. Each point on the y-axis represents an entry of the phase tracker's accumulator table. Each point on the graph corresponds to the value of the corresponding accumulator table entry at the end of a profiling interval. Dark values represent high execution frequency, while light values correspond to low frequency. The same trends that were seen in Figure 1 for `gzip` can be clearly seen in Figure 3. In both of these figures, when observing them at the coarsest granularity, we can see that there are at least three different phases labeled `A`, `B` and `C`. In Figure 3, the phase tracker table entries `2`, `5`, `7`, `13` and `17` distinguish the two identical long running phases labeled `A` from a group of three long running phases labeled `C`. Phase table entries `12` and `20` clearly distinguish phase `B` from both `A` and `C`. This figure is pictorial evidence that the phase tracker is able to break the program's execution into the corresponding phases based solely on the executed code, and that these phases correspond to the behavior seen across the different program metrics in Figure 1.

4.4 Forming a Footprint

After the profiling interval has elapsed, and branch block information has been accumulated, the phase must then be classified. To do this we keep a history of past phase information.

If we fix the number of instructions for a profiling interval, then we can divide each bucket by this fixed number to get the percentage of execution that was accounted for by all instructions mapped to that bucket. However, we do not need to know the exact percentages for each bucket. Instead of keeping the

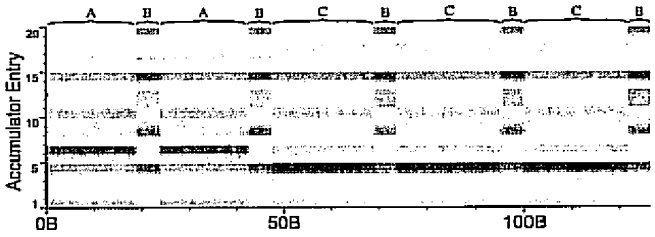

Figure 3: Visualization of the accumulator table used to track program behavior for `gzip`. *The x-axis is in billions of instructions, while the y-axis is the entry of the accumulator table. Each point on the graph corresponds to the value of the accumulator table at the end of a profiling interval where dark values correspond to more heavily accessed entries. The same trends that were seen in Figure 1 can be clearly seen in Figure 3.*

full counter values, we can instead compress phase information down to a couple of the most significant bits. This compressed information will then be kept in the Past Footprint table as shown in Figure 2.

The number of counter value bits that we need to observe is related to $Nbuckets$. As we increase the number of buckets, the data is spread over more buckets (table entries), making for less entries per bucket (better resolution) but at the cost of more area (both in terms of number of buckets and more bits per bucket). To be on the safe side, we would like *any* distribution of data into buckets to provide useful information. To achieve this we need to ensure that even if data is distributed perfectly evenly over all of the buckets, we would still record information about the frequency of those buckets. This can be achieved by reducing the accumulator counter by:

$$(bucket[i] \times Nbuckets)/(intervalsize)$$

If the number of buckets and interval size are powers of two, this is a simple shift operation. For the number of buckets we have chosen (32), and the interval size we profile over, this reduces the bucket size to 6 bits, and thus requires 24 bytes of storage for each unique phase in the Past Footprint table. In practice we see that the top 6 bits of the counter are more than enough to distinguish between two phases. In the worst case, you may need one or two more bits to reduce quantization error, but in reality we have not seen any programs that cause this to be an issue.

If too few buckets are used, aliasing effects can occur due to the hashing function, where two different phases will appear to have very similar Footprints. Therefore, we want to use a sufficiently large number of buckets to uniquely identify the differences in code execution between phases, while at the same time use only a small amount of area.

To examine the aliasing effect and determine what the appropriate number of buckets should be, Figure 4 shows the sum of the differences in the bucket weights found between all sequential intervals of execution. The y-axis shows the sum total of differences for each program. This is calculated by summing the

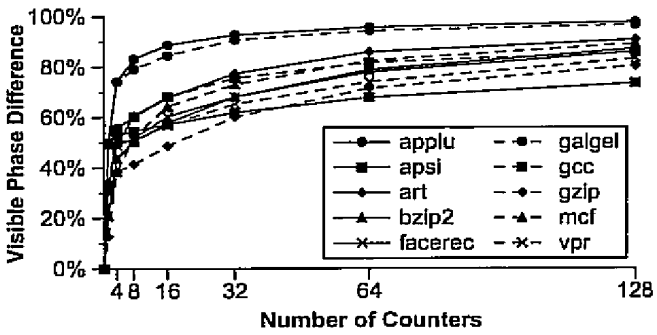

Figure 4: The percent difference found between Footprints from sequential intervals of execution, when varying the number of counters used to represent the footprints. The results are normalized to the difference between intervals found when having an infinite number of buckets to represent the footprint; 32 buckets captures most of the benefit.

differences between the buckets captured for interval i and $i-1$ for each interval i in the program. The x-axis is the number of distinct buckets used. All of the results are compared to the ideal case of using an infinite number of buckets (or one for each separate basic block) to create the Footprint. On the program `gcc` for example, the total sum of differences with 32 buckets was 72% of that captured with an infinite number of buckets. In general we have found that 32 buckets was enough to distinguish between two phases.

4.5 Classifying a Footprint to a Phase ID

After reducing the vector to form a footprint, we begin the classification process by comparing the footprint to a set of representative past footprint vectors. We compare the current vector to each vector in the table. The next section details how we perform the comparison and determine what a match is. If there is a match, we classify the profiled section of execution into the same phase as the past footprint vector, and the current vector is not inserted into the past footprint table. If there is no match, then we have just detected a new phase and hence must create a new unique phase ID into which we may classify it. This is done by choosing a unique phase ID out of a fixed pool of IDs. When allocating a new phase ID, we also allocate a new past footprint entry, set it to the current vector, and store with that entry the newly allocated phase ID. This allows future similar phases to be classified with the same ID. In this way only a single vector is kept for each unique phase ID, to serve as a representative of that phase. After a phase ID is provided for the most recent interval, it is passed along to prediction and statistic logging, and the phase identification part of our algorithm is completed.

To examine the number of phase IDs we need to track, Figure 5 shows the percentage of execution that can be accounted for by the top p phases, where p is shown on the x-axis. Results are graphed for the programs that had the min (`galgel`) and max (`art`) coverage, `gcc`, `gzip`, and the overall average. These results show that most of the program's phase behavior can be captured using a relatively small number of phase IDs.

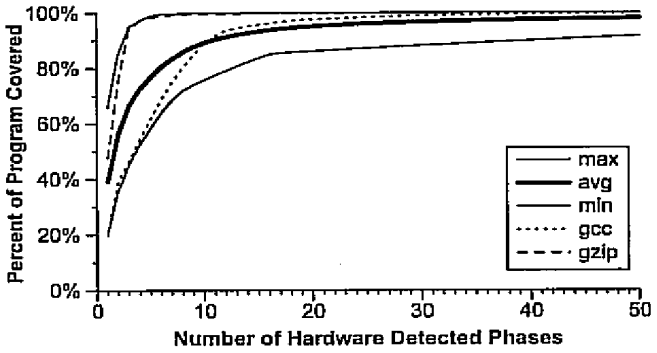

Figure 5: Results of the minimum number of phases that need to be captured versus the amount program execution they cover. The y-axis is the percent of program execution that is covered. The x-axis is the minimum number of phases needed to capture that much program execution.

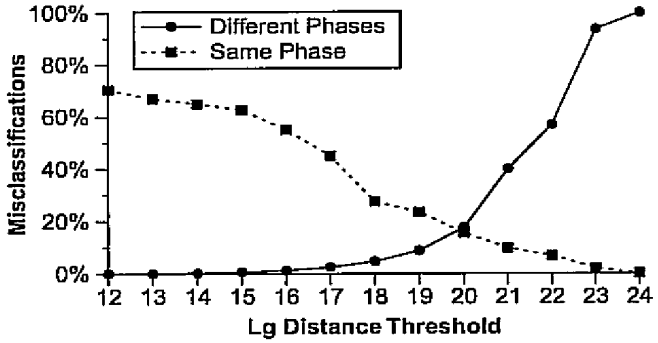

Figure 6: Results showing how well our hardware phase tracker classifies two sequential intervals of execution as being from "Different" or the "Same" phase of execution. The percent of misclassifications are shown in comparison to the phase classifications found using the off-line clustering SimPoint tool [21].

If we only track and optimize for the top 20 phases in each application, we will capture and be able to accurately apply phase prediction/optimizations to over 90% of the program's execution on average. In the worst case (min), we are able to optimize most of the program (over 80%) by only targeting a small number (20) of important recurring phases.

4.5.1 Finding a Match

We search through the Footprint histories to find a match, but this query is complicated by the fact that we are not necessarily searching for an exact match. Two sections of execution that have very similar footprints could easily be considered a match, even if they do not compare exactly. To compare two vectors to one another, we use the Manhattan distance between the two, which is the element-wise sum of the absolute differences. This distance is used to determine if the current interval should be classified as the same phase ID as one of the past footprint intervals.

If we set the distance threshold too low, the phase detection will be overly sensitive, and we will classify the program into many, very tiny phases which will cause us to lose any benefit from doing run-time phase analysis in the first place. If the threshold is too high, the classifier will not be able to distinguish between phases with different behavior. To quantify this effect, we examine how well our hardware technique classifies phases for a variety of thresholds compared to the phases found by the off-line clustering algorithm used in SimPoint [21].

The SimPoint tool is able to make global decisions to optimize the grouping of similar intervals into phases. The off-line algorithm makes no use of thresholds, instead its decisions are based solely on the structure found in the distribution of program behaviors. Our technique must be far more simplistic because it must be performed on-line and with limited computational overhead. This reduction in complexity comes at the cost of increased error.

The Different Phases line in Figure 6 shows the ability of our hardware technique to find phase changes (transitions between one phase and the next) when different thresholds are used to perform the phase classification. For example, when using a Manhattan distance of 1 million as our threshold (shown as 20 on our x-axis because it is in $\log_2$), our hardware technique identified 80% of the phase changes that occurred in the more complex off-line SimPoint analysis. Conversely, 20% of the phase changes were incorrectly classified as having the same phase ID as the last interval of execution.

Likewise, the Same Phases line in Figure 6 represents the ability of our hardware technique to accurately classify two sequential intervals as being part of the same phase as a function of different thresholds (again as compared to the off-line clustering analysis). For example, when using a Manhattan distance of 1 million (shown as 20 on the x-axis), our hardware technique identified 80% of the intervals that stayed in the same phase as correctly staying in the same phase, but 20% of those intervals were classified as having a different phase ID from the prior phase.

A misclassification occurs when two sequential intervals of execution are classified as being in the same phase or in different phases using our hardware approach when the off-line clustering analysis tool found the opposite for these two intervals.

If we are too aggressive and our hardware phase analysis indicates that there are phase changes when there are actually no noticeable changes in behavior, then we will create too many phase IDs that have similar behavior. This can create more overhead for performing phase-based optimization. On the other hand, if we are too passive in distinguishing between different phases, we will be missing opportunities to make phase specific optimizations.

In order to strike a balance between having a high capture rate and reducing the percent of false positives, we chose to use a threshold of 1 million. When comparing this with the interval size of 10 million instructions, this means that a difference in the phase behavior will be detected if 10% of the executed instructions are in different proportions. In choosing 1 million, we have on average a 20% misclassification rate. Note, that a misclassification does not necessarily mean that an incorrect optimization

will be performed. For example, if we have a "Same Phase" misclassification (the two intervals were really from the same phase, but were classified into different phases), then a phase change is observed using our hardware technique when there was not one in the baseline classifier. If the two hardware detected phases have the same optimization applied to them, then this misclassification can have no effect.

4.6 Per-Phase Performance Metric Homogeneity

Using the techniques presented above, we can perform phase classification on programs at run-time with little to no impact on the design of the processor core. One of the goals of phase classification is to divide the program into a set of phases that are fairly homogeneous. This means that an optimization adapted and applied to a single segment of execution from one phase, will apply equally well to the other parts of the phase. In order to quantify the extent to which we have achieved this goal, we need to test the homogeneity of a variety of architectural statistics on a per-phase basis.

Figure 7 shows the results of performing this analysis on the phases determined at run-time. Due to space constraints we only show results for two of the more complicated programs `gcc` and `gzip`. For both programs, a set of statistics for each phase is shown. The first phase that is listed (separated from the rest) as `full`, is the result of classifying the entire program into a single phase. The results show that for `gcc` for example, the average IPC of the entire program was 1.32, while the average number of cache misses was 445,083 per ten million instructions. In addition to just the average value, we also show the standard deviation for that statistic. For example, while the average IPC was 1.32 for `gcc`, it varied with a standard deviation of over 43% from interval to interval. If the phase-tracking hardware is successful in classifying the phases, the standard deviations for the various metrics should be low for a given phase ID.

Underneath the phase marked `full` are the five most frequently executed phases from the program as identified by our phase tracker. The phases are weighted by the percentage of the program's executed instructions they account for. For `gcc`, the largest phase accounts for 18.5% of the instructions in the entire program and has an average IPC of 0.61 and a standard deviation of only 1.6% (of 0.61). The other top four phases have standard deviations at or below this level, which means that our technique was successful at dividing up the execution of `gcc` into large phases with similar execution behavior with respect to IPC. Note, that some metrics for certain phases have a high standard deviation, but this occurs for architecture features/metrics that are unimportant for that phase. For example, the phase that occurs for 7.2% of execution in `gcc` has only 75 L1 instruction cache misses on average. This is an L1 miss rate of 0.00075%, so an error of 215% for this metric will not likely have any effect on the phase.

When we look at the energy consumption of `gcc`, it can be observed that energy consumption swings radically (a standard deviation of 90%) over the complete execution of the program. This can be seen visually in Figure 1, which plots the energy usage versus instructions executed. However, after dividing the program into phases, we see that each phase has very little variation within itself. All have less than 2% standard deviation. By analyzing `gcc` it can also be seen that the phase partitioning does a very good job across all of the measured statistics even though only *one* metric is used. This indicates that the phases that we have chosen are in some way representative of the actual behavior of the program.

5 Phase Prediction

The prior section described our phase tracking architecture, and how it can be used to classify phases. In this section we focus on using phase information to predict the next phase. For a variety of applications it is important to be able to predict future phase changes so that the system can configure for the code it will soon be executing rather than simply reacting to a change in behavior.

Figure 8 shows the percentage interval transitions that are changes in phase, for our set of benchmarks. For all of these programs, phase changes come quite often, but it should be noted that this statistic alone cannot gauge the complexity of the program behavior. The program `gcc` switches less than 10% of the time but switches between *many* different phases. The other extreme is `art` which switches almost half the time, but it is only switching between a few distinct phases. In this case, large repeating patterns can be observed. No two phases executing sequentially are that similar, but there is an order to the sequence. By adding in a prediction scheme for these cases, we not only take advantage of stable conditions as in past research, but actually take advantage of any repeating patterns in program behavior.

5.1 Markov Predictor

The prediction of phase behavior is different from many other systems in which hardware predictors are used. Because of this new environment, a new type of predictor has the potential to perform better than simply using predictors from other areas of computer architecture (branch and address prediction, memory disambiguation, etc.).

After observing the way that phases change, we determined that two pieces of information are important. First, the set of phases leading up to the prediction are very important, and second, the *duration* of execution of those phases is important.

A classic prediction model that is easily implementable in hardware is a Markov Model. Markov Models have been used in computer architecture to predict both prefetch addresses [13] and branches [8] in the past. The basic idea behind a Markov Model is that the next state of the system is related to the last set of states.

The intuition behind this design is that phase information tends to be characterized by many sections of stable behavior interspersed with abrupt phase changes. The key is to be able to predict when these phase changes will occur, and to know ahead of time what phase they will change to. The problem is that the changes are often preceded by stable conditions, and if we only consider the last couple of intervals we will not be able to tell the difference between sections of stable behavior that precede a phase change, and those sections that will continue to be stable. Instead, we need a way of compressing down stable phase

	phase	IPC	(stddev)	bpred	(stddev)	dl1	(stddev)	il1	(stddev)	energy	(stddev)	ul2	(stddev)
gcc	full	1.32	(43.4%)	27741	(135.5%)	445083	(110.7%)	50763	(203.2%)	6.44E+08	(90.0%)	227912	(139.7%)
	18.5%	0.61	(1.6%)	34665	(22.0%)	753382	(5.4%)	125091	(23.2%)	1.03E+09	(1.8%)	395997	(5.3%)
	18.1%	1.95	(0.3%)	13048	(3.9%)	28112	(15.1%)	43	(73.9%)	3.22E+08	(0.2%)	1006	(5.6%)
	7.2%	0.64	(0.2%)	843	(15.1%)	885081	(0.1%)	75	(215.5%)	9.78E+08	(0.3%)	443655	(0.1%)
	4.0%	1.49	(1.2%)	10145	(7.6%)	703554	(6.8%)	15591	(5.2%)	4.20E+08	(1.1%)	354084	(7.0%)
	3.9%	1.76	(1.6%)	2015	(13.6%)	98947	(5.9%)	102	(45.1%)	3.57E+08	(1.6%)	15595	(12.6%)
	phase	IPC	(stddev)	bpred	(stddev)	dl1	(stddev)	il1	(stddev)	energy	(stddev)	ul2	(stddev)
gzip	full	1.33	(16.3%)	56045	(11.1%)	90446	(58.2%)	60	(138.1%)	4.82E+08	(13.5%)	22880	(112.0%)
	17.1%	1.24	(3.4%)	53300	(10.8%)	96960	(10.1%)	12	(44.2%)	5.05E+08	(3.5%)	24218	(8.6%)
	9.4%	1.23	(3.8%)	54973	(11.5%)	99523	(11.3%)	11	(45.5%)	5.09E+08	(3.8%)	24518	(9.3%)
	8.8%	1.76	(0.6%)	56449	(4.8%)	37331	(5.6%)	241	(8.4%)	3.55E+08	(0.6%)	5617	(15.6%)
	8.0%	1.22	(4.3%)	54791	(6.8%)	99671	(11.9%)	40	(25.7%)	5.14E+08	(4.4%)	28153	(11.0%)
	7.4%	1.24	(3.1%)	55215	(11.1%)	96701	(9.6%)	12	(35.4%)	5.04E+08	(3.2%)	23701	(8.4%)

Figure 7: Examination of per-phase homogeneity compared to the program as a whole (denoted by `full`). For the two programs and each of the top 5 phases of each program, we show the average value of each metric and the standard deviation. The name of the phase is the percent of execution that it accounts for in terms of instructions. These results show that after dividing up the program into phases using our run-time scheme the behavior within each phase is quite consistent.

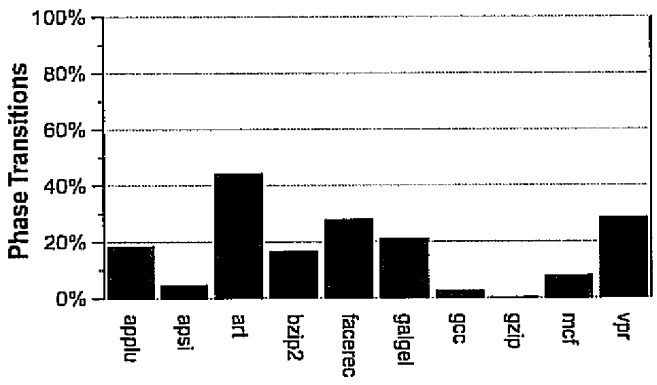

Figure 8: The percent of execution intervals that transition to a different phase from the prior execution interval's phase as found by our phase tracking architecture with 32 footprint counters using a 1 million Manhattan threshold.

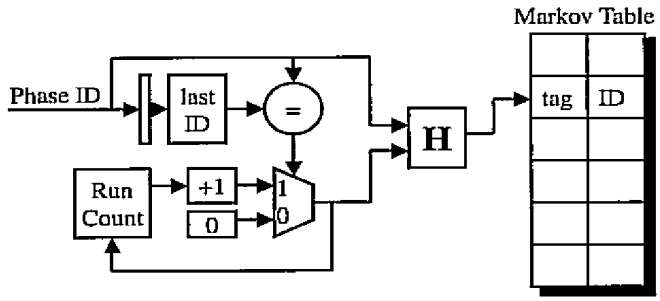

Figure 9: Phase Prediction Architecture for the Run Length Encoded (RLE) Markov predictor. The basic idea behind the predictor is that two pieces of information are used to generate the prediction, the phase id that was just seen, and the number of times prior to now that it has been seen in a row. The index into the prediction table is a hash of these two numbers.

information into a piece of information that we can use as state.

5.2 Run Length Encoding Markov Predictor

To compress the stable state we use a *Run Length Encoding* (RLE) Markov predictor. The basic idea behind the predictor is that it uses a run-length encoded version of the history to index into a prediction table. The index into the prediction table is a hash of the phase identifier and the number of times the phase identifier has occurred in a row.

Figure 9 shows our RLE Markov Phase ID prediction architecture. The the lower order bits of the hash function provide an index into the prediction table, and the higher order bits of the hash function provide a tag. When there is a tag match, the phase ID stored in the table provides a prediction as to the next phase to occur in execution. When there is a tag miss, the prior phase ID is assume to be the next phase ID to occur in the program's execution. We found that predicting the last phase ID to be 75% accurate on average.

We only update the predictor when there is (1) a change in the phase ID, or (2) when there is a tag match. We only insert an entry when there is a phase ID change, since we want to predict when the phase is going to change. Execution intervals where the same phase ID occurs several times in a row do not need to be stored in the table, since they will be correctly predicted as "last phase ID", when the there is a table miss. This helps table capacity constraints and avoids polluting the table with last phase predictions. For the second update case, when there is a tag match, we update the predictor because the observed run length may have potentially changed.

5.3 Predictor Comparison

We compare our RLE Markov phase predictor with other prediction schemes in Figure 10. This Figure has four bars for every program, and each bar corresponds to the prediction accuracy of a prediction architecture. The first and simplest scheme, `Last Phase`, simply predicts that the next phase is the same as the current phase, in essence always predicting stable operation. The prediction accuracy of this scheme is inversely proportional to the rate at which phases change in a given benchmark. For the program `gzip` for example, there are long periods of execution where the phase does not change, and therefore predicting no-change does exceptionally well.

In order to insure that we were not simply providing an

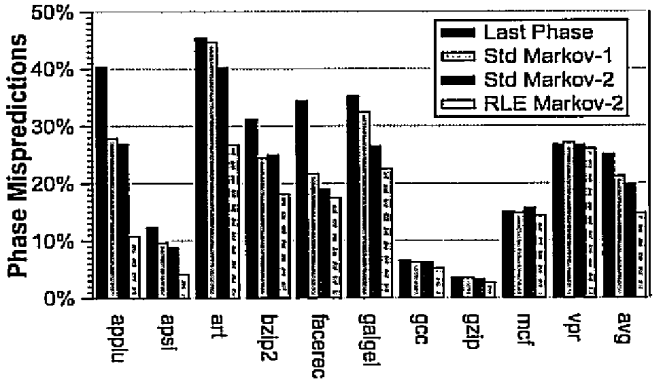

Figure 10: Phase ID Prediction Accuracy. This figure shows how well different prediction schemes work. The most naive scheme, `last`*, simply predicts that the phases never change. The bars marked* `Markov` *and* `RLE Markov` *show how well we can predict the phase identifiers if we use a Markov prediction scheme with a Markov table size of 256 entries.*

expensive filter for noise in the phase IDs, we also compared against a simple noise filter which works by predicting that the next phase will be the most commonly occurring of the last three phases seen. This is not shown, as it actually performed worse on all of the programs.

Additionally we wanted to examine the effect of a simple Markov model predictor for history lengths of 1 and 2. The Markov model predictor does a better job of predicting phase transitions than `Last Phase`, but it is limited by the fact that long runs will always be predicted as infinitely stable due to the history filling up. However, it is still very effective for `facerec` and `applu`, but does not provide much benefit for either `art` or `galgel`.

The final bar, `RLE Markov`, is our improved Markov predictor which compresses stable phases into a tuple of phase id and duration. All of the Markov predictors simulated had 256 entries taking up less than 500 bytes of storage. Using RLE Markov outperforms both the `Last Phase` and traditional `Markov` on all the benchmarks. It performs especially well compared to other schemes on both `applu` and `art`. Overall, using a Run-Length Encoded Markov predictor can cut the phase mispredictions down to 14% on average.

6 Applications

This section examines three optimization areas in which a phase-aware architecture can provide an advantage. We begin by examining the relationship between phase behavior and value locality. We then demonstrate ways to reduce processor energy consumption by adjusting the aggressiveness of the data cache and the instruction front end.

6.1 Frequent Value Locality

Prior work on value predictors has shown that there is a great deal of value locality in a variety of programs [14, 7]. Recently, researchers have started to take advantage of frequently loaded values for the purpose of optimizing caches. For example, Yang and Gupta [22] proposed a data cache organization that compresses the most frequently used program values in order to save energy. Another way of exploiting value locality is through value specialization, which can be done either statically or dynamically [6, 17, 16] to create specialized versions of procedures or code-regions based upon the values frequently seen. These techniques are built on the idea of finding the most frequent values for loads over the whole program, and then specializing the program to those frequent values.

We examine the potential of capturing frequent values on a per-phase basis and compare this to the frequent values aggregated over the entire program, as would be used in value code specialization [6]. To perform this experiment we first gathered the top 16 values that were loaded over the complete execution of the program and stored them into a table. We then examined the percentage of executed loads that found their loaded value in this table. This result is shown as `Static` in Figure 11. While significant portions of some programs are covered by just these few top values (such as `applu`), over half of the programs have less than 10% of their loaded values covered by these top values.

The question is: can we do better by exploiting hardware-detected phase information? To answer this question we take the top 16 values for each phase, as detected by the hardware phase tracker. These top values will be shared across a single phase even if it is split into two or more different sections of execution. Each load in the program is then checked against the top values for its corresponding phase. The `Phase Coverage` bar in Figure 11 shows the percent of all load values in the program that were successfully matched to it's per-phase top value set.

Without any notion of loads or values, our method of dividing up phases is very successful at assisting in the search for frequent values. By just tracking the top 16 values of each phase, we are able to capture the values from almost 50% of the executed loads on average. The `Perfect` bar shows percentage of loads covered if one captures the top 16 load values for each and every interval (i.e., 10 million instructions) separately. This is in effect the best that we could hope to achieve for an interval size of 10 million instructions, because the 16 entries in the value table are custom crafted for each interval individually. As shown in Figure 11, the phase-tracker compares favorably with the optimal coverage. Two thirds of the total possible benefit from per-interval value locality can be captured by per-phase value locality. It is important to point out this graph by itself is not a good indicator of usefulness as near perfect coverage could be achieved simply by making every interval a separate phase. However, as shown in Figure 5 only a few phases (around 20) are used to cover at least 80% of the program's execution.

6.2 Dynamic Data Cache Size Adaptation

In a modern processor a significant amount of energy is consumed by the data cache, but this energy may not be put to good use if an application is not accessing large amounts of data with high locality. To address this potential inefficiency, previous work has examined the potential of dynamically reconfiguring the data caches with the intention of saving power. In [2], Balasubramonian et. al. present two different schemes with

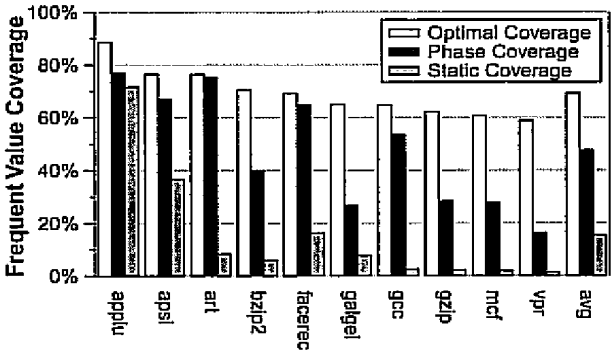

Figure 11: The percent of the program's load values that are found in a table of the most frequently values loaded over the whole program (Static Coverage), on a per-phase basis (Phase Coverage), and on a per execution interval basis (Optimal Coverage).

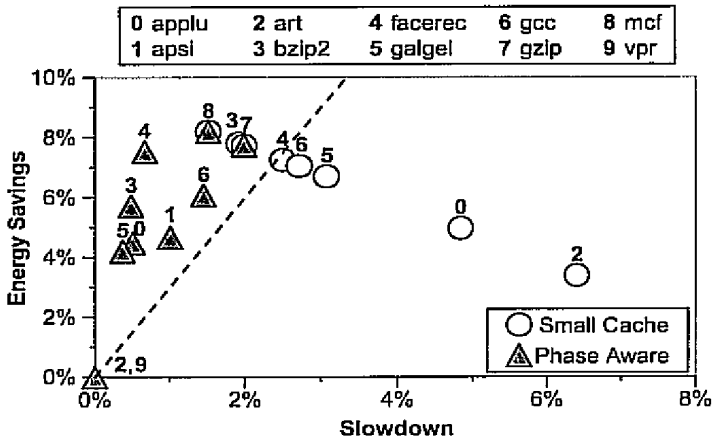

Figure 12: Data Cache Re-configuration. The tradeoff between energy savings and slowdown for two different cache policies. All results are relative to a 32KB 4-way associative cache. The circles in the graph (each labeled with a number for the program the data point is from) show the energy and performance of an 8KB direct mapped cache. The triangles show the tradeoff of intelligently switching between an 8KB direct mapped and a 32KB 4-way data cache based on phase classification and prediction.

which re-configuration may be guided. In one scheme, hardware performance counters are read by re-configuration software every hundred thousand cycles. The software then makes a decision based on the values of the counters. In another scheme, re-configuration decisions are performed on procedure boundaries instead of at fixed intervals. To reduce the overhead of re-configuration, software to trigger re-configuration is only placed before procedures that account for more than a certain percentage of execution.

Another form of re-configurable cache that has been proposed dynamically divides the data cache into multiple partitions, each of which can be used for a different function such as instruction reuse buffers, value predictors, etc [18]. These techniques can be triggered at different points in program execution including procedure boundaries and fixed intervals. The overhead of re-configuration can be quite large and making these policy decisions only when the large scale program behavior changes, as indicated by phase shifts in our hardware tracker, can minimize overhead while guaranteeing adequate sensitivity to attain maximum benefit.

We examined the use of phase tracking hardware to guide an energy aware, re-sizable cache. The energy consumption of the data cache can be reduced by dynamically shifting to a smaller, less associative cache configuration for program phases that do not benefit significantly from more aggressive cache configurations. By targeting only those phases that are predicted to have energy savings due to cache size reduction, our scheme is able to reduce power with very little impact on the performance.

We examined an architecture with two possible cache configurations, 32KB 4-way associative and 8KB direct mapped. In Figure 12, the trade off between these two configurations is plotted. For each program, we use the 32KB cache configuration as the baseline result. The labeled circles in Figure 12 show the total processor energy savings and performance degradation for each program if only the smaller (8KB) cache size is used. For example, a processor with a smaller cache configuration for the program `applu` is both 5% slower and uses 5% less energy. Two programs, `vpr` and `apsi`, actually use more energy with a smaller cache due to large slow downs. These two points are off the scale of this graph and are not shown.

While examining energy savings and slow down is interesting, it is important to note that there is more than one way to reduce both energy and performance. Voltage scaling in particular has proven to be a technology capable of reaping large energy savings for a relative reduction in performance. For our results, we assume that for voltage scaling a performance degradation of 5% will yield an approximate energy saving of 15%. We use this rule of thumb as our guideline for determining when to reduce the active size of the cache. In Figure 12, this simple model of voltage scaling is plotted as a dashed line. When the cache size is reduced, most programs fall far short of this baseline, meaning that voltage scaling would provide a better performance-energy tradeoff. There are a couple of exceptions, in particular `mcf`, `bzip`, and `gzip` do well even without any sort of phase-based re-configuration.

The shaded triangles in Figure 12 show what happens if we use phase classification and prediction to guide our re-configuration. When a new phase ID is seen, we sample the IPC and energy used for a few intervals using the 32KB 4-way cache, and a few intervals for the 8KB direct mapped cache. These samples could be kept in a small hardware profiling table associated with the phase ID. After taking these samples, if we find that a particular phase is able to achieve more than three times the energy savings relative to the slow down seen when using the 8KB cache, we then predict for this phase ID that the smaller cache size should be used. This heuristic means that the small cache size is used only if re-configuration would beat voltage scaling for that phase. After a decision has been made as to the con-

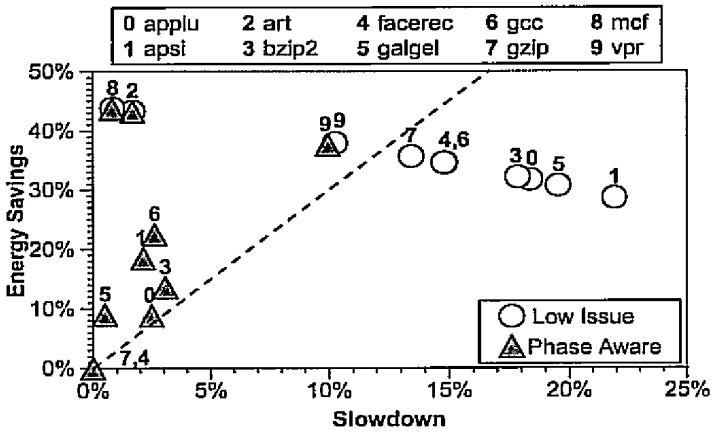

Figure 13: Processor Width Adaptation. The tradeoff between energy savings and slowdown for two different front end policies. All results are relative to an aggressive 8–issue machine. The circles in the graph (each labeled with a number for the program) show the energy and performance of a less aggressive 2–issue processor. The triangles show using the phase classifier and predictor for switching between 2–issue and 8–issue based on phase changes.

figuration to use for a phase ID, the corresponding cache size is stored in the phase profiling table/database associated with that phase ID. The phase classifier and predictor are then used to predict when a phase change occurs. When a phase change prediction occurs, the predicted phase ID looks up the cache size in the profiling table, and re-configures the cache (if it is not already that size) at the predicted phase change.

For all programs, our re-configuration is able to beat or tie voltage scaling. For example, using phase-based reconfiguration results in a slowdown of 0.5% for `applu`, while the total energy savings is 4.5%. Even the program `apsi`, which had *increased* energy consumption in the small cache configuration, is able to get almost 5% energy savings with only a 1% slowdown.

6.3 Dynamic Processor Width Adaptation

One way to reduce the energy consumption in a processor is to reduce the number of instructions entering the pipeline every cycle [12, 1]. We call this adjusting the width of the processor. Reducing the width of the processor reduces the demand on the fetch, decode, functional units, and issue logic. Certain phases can have a high degree of instruction level parallelism, whereas other phases have a very low degree. Take for example the top two phases for `gcc` shown in Figure 7. The intervals classified to be in the first phase consisting of 18.5% of execution have an IPC of 0.61 with a high data cache miss rate. In comparison, the intervals in the second most frequently encountered phase (accounting for 18.1% of execution) have an IPC of 1.95 and very low data cache miss rates. We can potentially save energy without hurting performance by throttling back the width of the processor for phases that have low IPC, while still using aggressive widths for phases with high IPC.

In the current literature, decisions to reduce or increase the fetch/decode/issue bandwidth of the processor are made either at fixed intervals (relatively short intervals such as 1,000 cycles) [12] or, as in the case of branch confidence based schemes, when a branch instruction is fetched [1]. It can very difficult to design real systems that save energy by reconfiguring at these speeds, but a hardware phase-tracker can help make these decisions at a coarser granularity while still maintaining performance and energy benefits.

We examined an architecture that could be configured with 2 different widths - one where up to 2 instructions are decoded and up to 2 issued per cycle, and one where up to 8 instructions are decoded and up to 8 issued per cycle. When a new phase ID is seen by the phase tracker, we sample the IPC for three intervals with a width of 2 instructions, and three intervals with a width of 8 instructions. If there is little difference in the IPC between these two widths, then we assign a width of 2 instructions to this Phase ID in our profiling table, otherwise we assign a width of 8 instructions. During execution, we use the phase ID predictor to effectively predict the width for the next interval of execution and adjust the processor's width accordingly. Our results show that the chosen configuration for a given phase can be trained (1) with only a few samples, and (2) only once to accurately represent the behavior of a given phase ID. This requires very little training time due to the fact that 20 or fewer phase IDs are needed to capture 80% or more of a program's execution as shown in Figure 5.

Figure 13 is a graph of the results seen when applying phase-directed width re-configuration. The white circles in the graph show the behavior of running the programs on only a 2–wide machine relative to the more aggressive 8–wide machine. The dotted line again shows what could potentially be achieved if voltage scaling was used. While `mcf` and `art` save a lot of energy with little performance degradation on a 2–wide machine, the other programs do not fair as well. The program `apsi`, for example, has a slowdown of over 22% with an energy savings of around 30%. This does not compare favorably to voltage scaling (as discussed in Section 6.2). On the other hand if we use phase-directed width throttling on `apsi`, a total processor energy savings of 18% can be achieved with only 2.2% slowdown.

For all of the programs we examined, with one exception, the slowdown due to phase aware width throttling was less than 4%, while the average energy savings was 19.6%. This result demonstrates that there is significant benefit to be had in the reconfiguration of processor front end resources even at very large granularities. In the worst case, this will mean a re-configuration every 10 million instructions, and on average every 70 million instructions. This should be designable even under conservative assumptions.

7 Summary

In this paper we present an efficient run-time phase tracking architecture that is based on detecting changes in the code being executed. This is accomplished by dividing up all instructions seen into a set of buckets based on branch PCs. This way we approximate the effect of taking a random projection of the basic

block vector, which was shown in [21] to be an effective method of identifying phases in programs.

Using our phase classification architecture with less than 500 bytes of on-chip memory, we show that for most programs, a significant amount of the program (over 80%) is covered by 20 or less distinct phases. Furthermore, we show that these phases, while being distinct from one another, have fairly uniform behavior within a phase, meaning that most optimizations applied to one phase will work well on all intervals in that phase. In the program `gcc`, the IPC attained by the processor on average over the full run of execution is 1.32, but has a standard deviation of more than 43%. By dividing it up into different phases, we achieve much more stable behavior, with IPCs ranging between 0.61 and 1.95, but now with standard deviations of less than 2%.

In addition to this, we present a novel phase prediction architecture using a Run Length Encoding Markov predictor that can predict not only when a phase change is about to occur, but to which phase ID it will transition to. In using this design, which also uses less than 500 bytes of storage, we achieve a phase prediction miss rate of 10% for `applu` and 4% for `apsi`. In comparison, always predicting that the phase will stay the same results in a miss rate of 40% and 12% respectively.

We also examined using our phase tracking and prediction architecture to enable new phase-directed optimizations. Traditional architecture and software optimizations are targeted at the average or aggregate behavior of a program. In comparison, phase-directed optimizations aim at optimizing a program's performance tailored to the different phases in a program. In this paper, we examined using phase tracking and prediction to increase frequent value profiling coverage, and to provide energy savings through data cache and processor width re-configuration.

We believe our phase tracking and prediction design will open the door for a new class of run-time optimization that targets large scale program behavior. Even though we present a hardware implementation for phase tracking, a similar design can be implemented in software to perform phase classification for run-time optimizers, just-in-time compilation systems, and operating systems. Hardware and software optimizations that can potentially benefit the most from phase classification and prediction are (1) those that need expensive profiling/training before applying an optimization, (2) those where the time or cost it takes to perform the optimization is either slow or expensive, and (3) those that can benefit from specialization where they have the same code/data being used differently in different phases of execution. By using our dynamic phase tracking and prediction design, phase-behavior can be characterized and predicted at the largest of scales, providing a unified mechanism for phase-directed optimization.

Acknowledgments

We would like to thank Jeremy Lau and the anonymous reviewers for providing useful comments on this paper. This work was funded in part by NSF CAREER grant No. CCR-9733278, Semiconductor Research Corporation grant No. SRC-2001-HJ-897, and an equipment grant from Intel.

References

[1] J.L. Aragon, J. Gonzalez, and A. Gonzalez. Power-aware control speculation through selective throttling. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture*, February 2003.

[2] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *33rd International Symposium on Microarchitecture*, pages 245–257, 2000.

[3] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W. W. Hwu. Vacuum packing: Extracting hardware-detected program phases for post-link optimization. In *35th International Symposium on Microarchitecture*, December 2002.

[4] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.

[5] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, U. of Wisconsin, Madison, June 1997.

[6] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, March 1999.

[7] B. Calder, G. Reinman, and D.M. Tullsen. Selective value prediction. In *26th Annual International Symposium on Computer Architecture*, pages 64–74, June 1999.

[8] I.-C. Chen, J. T. Coffey, and T. N. Mudge. Analysis of branch prediction via data compression. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 128–137, October 1996.

[9] A. Dhodapkar and J. E. Smith. Dynamic microarchitecture adaptation via co-designed virtual machines. In *International Solid State Circuits Conference*, February 2002.

[10] A. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *29th Annual International Symposium on Computer Architecture*, May 2002.

[11] M. Huang, J. Renau, and J. Torrellas. Profile-based energy reduction in high-performance processors. In *4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2001.

[12] A. Iyer and D. Marculescu. Power aware microarchitecture resource scaling. In *Proceedings of the DATE 2001 on Design, automation and test in Europe*, pages 190–196, 2001.

[13] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[14] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. Value locality and load value prediction. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, October 1996.

[15] M. Merten, A. Trick, R. Barnes, E. Nystrom, C. George, J. Gyllenhaal, and Wen mei W. Hwu. An architectural framework for run-time optimization. *IEEE Transactions on Computers*, 50(6):567–589, June 2001.

[16] M. Mock, C. Chambers, and S.J. Eggers. Calpa: a tool for automating selective dynamic compilation. In *33rd International Symposium on Microarchitecture*, pages 291–302, December 2000.

[17] R. Muth, S.A. Watterson, and S.K. Debray. Code specialization based on value profiles. In *Static Analysis Symposium*, pages 340–359, 2000.

[18] P. Ranganathan, S. V. Adve, and N.P. Jouppi. Reconfigurable caches and their application to media processing. In *27th Annual International Symposium on Computer Architecture*, pages 214–224, June 2000.

[19] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report UCSD-CS99-630, UC San Diego, August 1999.

[20] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[21] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[22] J. Yang and R. Gupta. Frequent value locality and its applications. *Special Issue on Memory Systems, ACM Transactions on Embedded Computing Systems*, 1(1):79–105, November 2002.

EXHIBIT 5
TO DECLARATION OF
JAMES A FLIGHT

From: Tim Sherwood [sherwood@cs.ucsb.edu]
Sent: Monday, October 30, 2006 4:28 PM
To: James A. Flight
Subject: Re: Phase Tracking and Prediction publication

Hi Jim,

The publication appeared in ISCA 2003 and so was officially published on June 9th, 2003.

http://cs.nyu.edu/isca03/

-Tim

On Mon, 30 Oct 2006, James A. Flight wrote:
> I am trying to properly cite to your publication "Timothy Sherwood,
> Suleyman Sair, and Brad Calder.
> <http://www-cse.ucsd.edu/~calder/abstracts/ISCA-03-Phase.html> Phase
> Tracking and Prediction, In the proceedings of the 30th Annual Intl.
> Symposium on Computer Architecture (ISCA 2003), June 2003. San Diego,
> California."
>
>
>
> I ran across this link
> "http://www-cse.ucsd.edu/Dienst/UI/2.0/Describe/ncstrl.ucsd_cse/CS2002-0710"
> indicates that you actually published this one year earlier (i.e., on
> June 23, 2002). Can you please confirm which is the correct date?
>
>
>
> Thank you very much,
>
>
>
> Jim
>
>
>
> James A. Flight
>
>
>
> 20 North Wacker Drive, Suite 4220
> Chicago, Illinois 60606
>
> (312) 580-1034 (Direct)
> (312) 580-1020 (Main)
> (312) 580-9696 (Fax)
>
> jflight@hfzlaw.com
>
>
>
>
>
>
> Important: This electronic mail message and any attached files contain
> information intended for the exclusive use of the individual or entity
> to whom it is addressed and may contain information that is
> proprietary, privileged, confidential and/or exempt from disclosure
> under applicable law. If you are not the intended recipient, please
> notify the sender, by electronic mail or telephone, of any unintended

> recipients and delete the original message without making any copies.
>
> >
> >
>

EXHIBIT 6
TO DECLARATION OF
JAMES A FLIGHT

ISCA 2003

The thirtieth International Symposium on Computer Architecture (ISCA) will be held at the Town and Country Hotel in San Diego 9-11 June, 2003. ISCA 2003 is a constitutent conference in the ACM Federated Computing Research Conference (FCRC).

Main Page

Final Program

Travel and Registration

Student Travel Grants

Companion Travel Grants

Workshops and Tutorials

Call for Papers

Organizing Committee

Program Committee

Allan Gottlieb

EXHIBIT 7
TO DECLARATION OF
JAMES A FLIGHT

Phase tracking and prediction

Full text Pdf (674 KB)

Source **ACM SIGARCH Computer Architecture News** archive
Volume 31 , Issue 2 (May 2003) table of contents
SESSION: Prediction table of contents
Pages: 336 - 349
Year of Publication: 2003
ISSN:0163-5964
Also published in ...

Authors Timothy Sherwood University of California, San Diego
Suleyman Sair University of California, San Diego
Brad Calder University of California, San Diego

Publisher ACM Press New York, NY, USA

↑ ABSTRACT

In a single second a modern processor can execute billions of instructions. Obtaining a bird's eye view of the behavior of a program at these speeds can be a difficult task when all that is available is cycle by cycle examination. In many programs, behavior is anything but steady state, and understanding the patterns of behavior, at run-time, can unlock a multitude of optimization opportunities.In this paper, we present a unified profiling architecture that can efficiently capture, classify, and predict phase-based program behavior on the largest of time scales. By examining the proportion of instructions that were executed from different sections of code, we can find generic phases that correspond to changes in behavior across many metrics. By classifying phases generically, we avoid the need to identify phases for each optimization, and enable a unified prediction scheme that can forecast future behavior. Our analysis shows that our design can capture phases that account for over 80% of execution using less that 500 bytes of on-chip memory.

↑ REFERENCES

Note: OCR errors may be found in this Reference List extracted from the full text article. ACM has opted to expose the complete List rather than only correct and linked references.

1 Juan L. Aragón , José González , Antonio González, Power-Aware Control Speculation through Selective Throttling, Proceedings of the 9th International Symposium on High-Performance Computer Architecture, p.103, February 08-12, 2003

2 Rajeev Balasubramonian , David Albonesi , Alper Buyuktosunoglu , Sandhya Dwarkadas, Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures, Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture, p.245-257, December 2000, Monterey, California, United States [doi>10.1145/360128.360153]

3 Ronald D. Barnes , Erik M. Nystrom , Matthew C. Merten , Wen-mei W. Hwu, Vacuum packing: extracting hardware-detected program phases for post-link optimization, Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture, November 18-22, 2002, Istanbul, Turkey

4 David Brooks , Vivek Tiwari , Margaret Martonosi, Wattch: a framework for architectural-level power analysis and optimizations, Proceedings of the 27th annual international symposium on Computer architecture, p.83-94, June 2000, Vancouver, British Columbia, Canada [doi>10.1145/342001.339657]

5 D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, U. of Wisconsin, Madison, June 1997.

6 B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. Journal of Instruction Level Parallelism, March 1999.

7 Brad Calder , Glenn Reinman , Dean M. Tullsen, Selective value prediction, Proceedings of the 26th annual international symposium on Computer architecture, p.64-74, May 01-04, 1999, Atlanta, Georgia, United States [doi>10.1145/307338.300985]

8 I-Cheng K. Chen , John T. Coffey , Trevor N. Mudge, Analysis of branch prediction via data compression, Proceedings of the seventh international conference on Architectural support for programming languages and operating systems, p.128-137, October 01-04, 1996, Cambridge, Massachusetts, United States [doi>10.1145/248208.237171]

9 A. Dhodapkar and J. E. Smith. Dynamic microarchitecture adaptation via co-designed virtual machines. In International Solid State Circuits Conference, February 2002.

10 Ashutosh S. Dhodapkar , James E. Smith, Managing multi-configuration hardware via dynamic working set analysis, Proceedings of the 29th annual international symposium on Computer architecture, p.233, May 25-29, 2002, Anchorage, Alaska [doi>10.1145/545214.545241]

11 M. Huang, J. Renau, and J. Torrellas. Profile-based energy reduction in high-performance processors. In 4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4), December 2001.

12 A. Iyer , D. Marculescu, Power aware microarchitecture resource scaling, Proceedings of the conference on Design, automation and test in Europe, p.190-196, March 2001, Munich, Germany

13 Doug Joseph , Dirk Grunwald, Prefetching using Markov predictors, Proceedings of the 24th annual international symposium on Computer architecture, p.252-263, June 01-04, 1997, Denver, Colorado, United States [doi>10.1145/384286.264207]

14 Mikko H. Lipasti , Christopher B. Wilkerson , John Paul Shen, Value locality and load value prediction, Proceedings of the seventh international conference on Architectural support for programming languages and operating systems, p.138-147, October 01-04, 1996, Cambridge, Massachusetts, United States [doi>10.1145/248208.237173]

15 Matthew C. Merten , Andrew R. Trick , Ronald D. Barnes, An Architectural Framework for Runtime Optimization, IEEE Transactions on Computers, v.50 n.6, p.567-589, June 2001 [doi>10.1109/12.931894]

16 Markus Mock , Craig Chambers , Susan J. Eggers, Calpa: a tool for automating selective dynamic compilation, Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture, p.291-302, December 2000, Monterey, California, United States [doi>10.1145/360128.360158]

17 Robert Muth , Scott A. Watterson , Saumya K. Debray, Code Specialization Based on Value Profiles, Proceedings of the 7th International Symposium on Static Analysis, p.340-359, June 29-July 01, 2000

18 Parthasarathy Ranganathan , Sarita Adve , Norman P. Jouppi, Reconfigurable caches and their application to media processing, Proceedings of the 27th annual international symposium on Computer architecture, p.214-224, June 2000, Vancouver, British Columbia, Canada [doi>10.1145/339647.339685]

19 T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report UCSD-CS99-630, UC San Diego, August 1999.

20 Timothy Sherwood , Erez Perelman , Brad Calder, Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications, Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques, p.3-14, September 08-12, 2001

21 Timothy Sherwood , Erez Perelman , Greg Hamerly , Brad Calder, Automatically characterizing large scale program behavior, Proceedings of the 10th international conference on Architectural support for programming languages and operating systems, October 05-09, 2002, San Jose, California [doi>10.1145/605432.605403]

22 Jun Yang , Rajiv Gupta, Frequent value locality and its applications, ACM Transactions on Embedded Computing Systems (TECS), v.1 n.1, p.79-105, November 2002 [doi>10.1145/581888.581894]

↑ **CITED BY** ***27***

Wonbok Lee , Kimish Patel , Massoud Pedram, B²Sim:: a fast micro-architecture simulator based on basic block characterization, Proceedings of the 4th international conference on Hardware/software codesign and system synthesis, October 22-25, 2006, Seoul, Korea

Chang-Burm Cho , Tao Li, Complexity-based program phase analysis and classification, Proceedings of the 15th international conference on Parallel architectures and compilation techniques, September 16-20, 2006, Seattle, Washington, USA

Kaushal Sanghai , Ting Su , Jennifer Dy , David Kaeli, A multinomial clustering model for fast simulation of computer architecture designs, Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining, August 21-24, 2005, Chicago, Illinois, USA

Michela Becchi , Patrick Crowley, Dynamic thread assignment on heterogeneous multiprocessor architectures, Proceedings of the 3rd conference on Computing frontiers, May 03-05, 2006, Ischia, Italy

Steven P. Reiss, Dynamic detection and visualization of software phases, ACM SIGSOFT Software Engineering Notes, v.30 n.4, July 2005

Priya Nagpurkar , Chandra Krintz, Visualization and analysis of phased behavior in Java programs, Proceedings of the 3rd international symposium on Principles and practice of programming in Java, June 16-18, 2004, Las Vegas, Nevada

Radu Cornea , Alex Nicolau , Nikil Dutt, Software annotations for power optimization on mobile devices, Proceedings of the conference on Design, automation and test in Europe: Proceedings, March 06-10, 2006, Munich, Germany

- Anahita Shayesteh , Glenn Reinman , Norman Jouppi , Suleyman Sair , Tim Sherwood, Dynamically configurable shared CMP helper engines for improved performance, ACM SIGARCH Computer Architecture News, v.33 n.4, November 2005
- Anahita Shayesteh , Glenn Reinman , Norm Jouppi , Tim Sherwood , Suleyman Sair, Improving the performance and power efficiency of shared helpers in CMPs, Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems, October 22-25, 2006, Seoul, Korea
- Cristiano Pereira , Jeremy Lau , Brad Calder , Rajesh Gupta, Dynamic phase analysis for cycle-close trace generation, Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, September 19-21, 2005, Jersey City, NJ, USA
- Chen Ding , Chengliang Zhang , Xipeng Shen , Mitsunori Ogihara, Gated memory control for memory monitoring, leak detection and garbage collection, Proceedings of the 2005 workshop on Memory system performance, June 12-12, 2005, Chicago, Illinois
- John D. Davis , Cong Fu , James Laudon, The RASE (Rapid, Accurate Simulation Environment) for chip multiprocessors, ACM SIGARCH Computer Architecture News, v.33 n.4, November 2005
- Kartik K. Agaram , Stephen W. Keckler , Calvin Lin , Kathryn S. McKinley, Decomposing memory performance: data structures and phases, Proceedings of the 2006 international symposium on Memory management, June 10-11, 2006, Ottawa, Ontario, Canada
- Domingo Benitez , Juan C. Moure , Dolores I. Rexachs , Emilio Luque, Evaluation of the field-programmable cache: performance and energy consumption, Proceedings of the 3rd conference on Computing frontiers, May 03-05, 2006, Ischia, Italy

  Priya Nagpurkar , Chandra Krintz, Phase-based visualization and analysis of Java programs, Science of Computer Programming, v.59 n.1-2, p.64-81, January 2006
- Ted Huffmire , Tim Sherwood, Wavelet-based phase classification, Proceedings of the 15th international conference on Parallel architectures and compilation techniques, September 16-20, 2006, Seattle, Washington, USA
- Tipp Moseley , Alex Shye , Vijay Janapa Reddi , Matthew Iyer , Dan Fay , David Hodgdon , Joshua L. Kihm , Alex Settle , Dirk Grunwald , Daniel A. Connors, Dynamic run-time architecture techniques for enabling continuous optimization, Proceedings of the 2nd conference on Computing frontiers, May 04-06, 2005, Ischia, Italy
- Thomas Y. Yeh , Glenn Reinman, Fast and fair: data-stream quality of service, Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems, September 24-27, 2005, San Francisco, California, USA
- Andy Georges , Dries Buytaert , Lieven Eeckhout , Koen De Bosschere, Method-level phase behavior in java workloads, ACM SIGPLAN Notices, v.39 n.10, October 2004
- Juan C. Moure , Domingo Benítez , Dolores I. Rexachs , Emilio Luque, Wide and efficient trace prediction using the local trace predictor, Proceedings of the 20th annual international conference on Supercomputing, June 28-July 01, 2006, Cairns, Queensland, Australia
- Trishul M. Chilimbi , Vinod Ganapathy, HeapMD: identifying heap-based bugs using anomaly detection, ACM SIGARCH Computer Architecture News, v.34 n.5, December 2006
- Xipeng Shen , Yutao Zhong , Chen Ding, Locality phase prediction, ACM SIGOPS Operating Systems Review, v.38 n.5, December 2004
- Xiaodong Li , Zhenmin Li , Francis David , Pin Zhou , Yuanyuan Zhou , Sarita Adve , Sanjeev Kumar, Performance directed energy management for main memory and disks, ACM SIGARCH

Computer Architecture News, v.32 n.5, December 2004

Xiaodong Li , Zhenmin Li , Yuanyuan Zhou , Sarita Adve, Performance directed energy management for main memory and disks, ACM Transactions on Storage (TOS), v.1 n.3, p.346-380, August 2005

Shiwen Hu , Madhavi Valluri , Lizy Kurian John, Effective management of multiple configurable units using dynamic optimization, ACM Transactions on Architecture and Code Optimization (TACO), v.3 n.4, p.477-501, December 2006

Priya Nagpurkar , Hussam Mousa , Chandra Krintz , Timothy Sherwood, Efficient remote profiling for resource-constrained devices, ACM Transactions on Architecture and Code Optimization (TACO), v.3 n.1, p.35-66, March 2006

Jedidiah R. Crandall , Gary Wassermann , Daniela A. S. de Oliveira , Zhendong Su , S. Felix Wu , Frederic T. Chong, Temporal search: detecting hidden malware timebombs with virtual machines, ACM SIGPLAN Notices, v.41 n.11, November 2006

↑ Collaborative Colleagues:

Brad Calder:	Matthew Arnold Todd Austin Todd M. Austin Jean-Loup Baer Iris Bahar Karan Bhatia Trevor Blackwell Larry Carter Lori Carter Andrew Chien Andrew A. Chien Trishul Chilimbi Weihaw Chuang Chandra Ckrintz Robert Cohn Osvaldo Colavin Jean-Francois Collard Jamison Collins Lieven Eeckhout Stephen Elbert Joel Emer Alan Eustace Peter Feller Jeanne Ferrante	Nikolas Gloy Ruben Gonzalez David Grove Dirk Grunwald Rajesh Gupta Urs Hölzle Greg Hamerly Amir H. Hashemi Michael Hind Yuanfang Hu Lizy K. John Simmi John Michael Jones Norman P. Jouppi David R. Kaeli Barbara Kreaseck Chandra Krintz Rakesh Kumar Jeremy Lau Dennis Lee Han Bok Lee Derek Lieber David J. Lilja Donald Lindsay	James Martin Michael Mozer Satish Narayanasamy Mark Oskin Harish Patil Cristiano Pereira Erez Perelman Gilles Pokam Gilles Pokam Gilles Pokam Glenn Reinman Glenn D. Reinman Suleyman Sair Jack Sampson Jack Sampson Vivek Sarkar Mike Schlansker Stefan Schoenmackers John Shen Timothy Sherwood Timothy Peter Sherwood Tomothy Sherwood Beth Simon James E. Smith	Michael D. Smith Amitabh Srivastava Nathan Tuck Dean Tullsen Dean M. Tullsen Eric Tune Gary Tyson Michael Van Biesbrouck Michael Van Biesbrouk George Varghese Ganesh Venkatesh Steven Wallace Hong Wang Ju Wang Perry Wang Don Yang Joshua J. Yi Weifeng Zhang Weifeng Zhang Ben Zorn Benjamin Zorn Benjamin G. Zorn
Suleyman Sair:	Brad Calder Jamison Collins Jose Fridman Yuanfang Hu Norm Jouppi Norman Jouppi David Kaeli Youngsoo Kim Satish Narayanasamy Guiseppe Olivadoti	Glenn Reinman Glenn Reinman Anahita Shayesteh Tim Sherwood Timothy Sherwood Dean M. Tullsen George Varghese		
Timothy	Banit Agrawal	Ryan Kastner	Mark Oskin	

Sherwood:	Forrest Brewer	Chandra Krintz	Erez Perelman
	Brett Brotherton	Jeremy Lau	Suleyman Sair
	Andrew P. Brown	Hua Lee	Stefan
	Brad Calder	Yan Meng	Schoenmackers
	Frederic T. Chong	Yan Meng	Lin Tan
	Joel Emer	Farilee Mintz	Michael Van
	Greg Hamerly	Hussam Mousa	Biesbrouck
	Greg Hoover	Priya Nagpurkar	Michael Van
	Ronald A. Iltis	Satish	Biesbrouk
		Narayanasamy	George Varghese
			Miroslava Vomela

Peer to Peer - Readers of this Article have also read:

- Data structures for quadtree approximation and compression **Communications of the ACM** 28, 9
  Hanan Samet
- A hierarchical single-key-lock access control using the Chinese remainder theorem **Proceedings of the 1992 ACM/SIGAPP Symposium on Applied computing**
  Kim S. Lee , Huizhu Lu , D. D. Fisher
- The GemStone object database management system **Communications of the ACM** 34, 10
  Paul Butterworth , Allen Otis , Jacob Stein
- Putting Innovation to work: adoption strategies for multimedia communication systems **Communications of the ACM** 34, 12
  Ellen Francik , Susan Ehrlich Rudman , Donna Cooper , Stephen Levine
- An intelligent component database for behavioral synthesis **Proceedings of the 27th ACM/IEEE conference on Design automation**
  Gwo-Dong Chen , Daniel D. Gajski

This Article has also been published in:

- **International Symposium on Computer Architecture**
  Proceedings of the 30th annual international symposium on Computer architecture
  2003 , San Diego, California

EXHIBIT 8
TO DECLARATION OF
JAMES A FLIGHT

Phase Tracking and Prediction

Timothy Sherwood Suleyman Sair Brad Calder

Department of Computer Science and Engineering
University of California, San Diego
{sherwood,ssair,calder}@cs.ucsd.edu

Abstract

In a single second a modern processor can execute billions of instructions. Obtaining a bird's eye view of the behavior of a program at these speeds can be a difficult task when all that is available is cycle by cycle examination. In many programs, behavior is anything but steady state, and understanding the patterns of behavior, at run-time, can unlock a multitude of optimization opportunities.

In this paper, we present a unified profiling architecture that can efficiently capture, classify, and predict phase-based program behavior on the largest of time scales. By examining the proportion of instructions that were executed from different sections of code, we can find generic phases that correspond to changes in behavior across many metrics. By classifying phases generically, we avoid the need to identify phases for each optimization, and enable a unified prediction scheme that can forecast future behavior. Our analysis shows that our design can capture phases that account for over 80% of execution using less that 500 bytes of on-chip memory.

1 Introduction

Modern processors can execute upwards of 5 billion instructions in a single second, yet most architectural features target program behavior on a time scale of hundreds to thousands of instructions, less than half a μS. While these optimizations can provide large benefits, they are limited in their ability to see the program behavior in a larger context.

Recently there has been a renewed interest in examining the run-time behavior of programs over longer periods of time [10, 11, 19, 20, 3]. It has been shown that programs can have considerably different behavior depending on which portion of execution is examined. More specifically, it has been shown that many programs execute as a series of phases, where each phase may be very different from the others, while still having a fairly homogeneous behavior within a phase. Taking advantage of this time varying behavior can lead to, among other things, improved power management, cache control, and more efficient simulation. The primary goal of this research is the development of a unified run-time phase detection and prediction mechanism that can be used to guide any optimization seeking to exploit large scale program behavior.

A phase of program behavior can be defined in several ways. Past definitions are built around the idea of a phase being an interval of execution during which a measured program metric is relatively stable. We extend this notion of a phase to include all similar sections of execution regardless of temporal adjacency. Simply put, if a phase of execution is correctly identified, there should only be small variations between any two execution intervals identified as being part of the same phase. A key point of this paper is that the phase behavior seen in any program metric is directly a function of the way the code is being executed. If we can accurately capture this behavior at run-time through the computation of a *single* metric, we can use this to guide many optimization and policy decisions without duplicating phase detection mechanisms for *each* optimization.

In this paper, we present an efficient run-time phase tracking architecture that is based on detecting changes in the *proportions* of the code being executed. In addition, we present a novel phase prediction architecture that can predict, not only when a phase change is about to occur, but also the phase to which it is will transition. Since our phase tracking implementation is based upon code execution frequencies, it is independent of any individual architecture metric. This allows our phase tracker to be used as a general profiling technique building up a profile or database of architecture information on a per phase basis to be used later for hardware or software optimization. Independence from architecture metrics allows us to consistently track phase information as the program's behavior changes due to phase-based optimizations.

We demonstrate the effectiveness of our hardware based phase detection and classification architecture at automatically partitioning the behavior of the program into homogeneous phases of execution and to identify phase changes. We show that the changes in many important metrics, such as IPC and energy, correlate very closely with the phase changes found by our metric. We then evaluate the effectiveness of phase tracking and prediction for value profiling, data cache reconfiguration, and re-configuring the width of the processor.

The rest of the paper is laid out as follows. In Section 2, prior work related to phase-based program behavior is discussed. Simulation methodology and benchmark descriptions can be found in Section 3. Section 4 describes our phase tracking architecture. The design and evaluation of the phase predictor are found in Section 5. Section 6 presents several potential applications of our phase tracking architecture. Finally, the results are summarized in Section 7.

2 Related Work

In this Section we describe work related to phase identification and phase-based optimization.

In [19], we provided an initial study into the time varying behavior of programs, showing that programs have repeatable phase-based behavior over many hardware metrics – cache behavior, branch prediction, value prediction, address prediction,

IPC and RUU occupancy for all the SPEC 95 programs. Looking at these metrics over time, we found that many programs have repeating patterns, and that important metrics tend to change at the same time. These places represent phase boundaries.

In [20], we proposed that by profiling only the code that was executed over time we could automatically identify periodic and phase behavior in programs. The goal was to automatically find the repeating patterns observed in [19], and the lengths (periods) of these patterns. We then extended this work in [21], using techniques from machine learning to break the complete execution of the program into phases (clusters) by only tracking the code executed. We found that intervals of execution grouped into the same phase had similar behavior across all the architecture metrics examined. From this analysis, we created a tool called SimPoint [21], which automatically identifies a small set of intervals of execution (simulation points) in a program to perform architecture simulations. These simulation points provide an accurate and efficient representation of the complete execution of the program.

The work of Dhodapkar and Smith [10, 9] is the most closely related to ours. They found a relationship between phases and instruction working sets, and that phase changes occur when the working set changes. They propose that by detecting phases and phase changes, multi-configuration units can be re-configured in response to these phase changes. They have used their working set analysis for instruction cache, data cache and branch predictor re-configuration to save energy [10, 9].

The work we present in this paper identifies phases and phase changes by keeping track of the proportions in which the code was executed during an interval based upon the profiler used in [20]. In comparison, Dhodapkar and Smith [10, 9] track the phase and phase changes solely upon what code was executed (working set), without weighting the code by its frequency of execution. Future research is needed to compare these two approaches.

Additional differences between our work include our examination of architectures for predicting phase changes, and different uses from [10, 9], such as value profiling and processor width reconfiguration. We provide an architecture that can fairly accurately predict what the next phase will be, along with predicting when there will be a phase change. In comparison, Dhodapkar and Smith do not examine phase-based prediction [10, 9], but concentrate on detecting when the working set size changes, and then reactively apply optimization.

Merten et al. [15] developed a run-time system for dynamically optimizing frequently executed code. Then in [3], Barnes et al. extend this idea to perform phase-directed complier optimizations. The main idea is the creation of optimized code "packages" that are targeted towards a given phase, with the goal of execution staying within the package for that phase. Barnes et al. concentrate primarily on the compiler techniques needed to make phase-directed compiler optimizations a reality, and do not examine the mechanics of hardware phase detection and classification. We believe that using the techniques in [3] in conjunction with our phase classification and prediction architecture will provide a powerful run-time execution environment.

I Cache	16k 4-way set-associative, 32 byte blocks, 1 cycle latency
D Cache	16k 4-way set-associative, 32 byte blocks, 1 cycle latency
L2 Cache	128K 8-way set-associative, 64 byte blocks, 12 cycle latency
Main Memory	120 cycle latency
Branch Pred	hybrid - 8-bit gshare w/ 2k 2-bit predictors + a 8k bimodal predictor
O-O-O Issue	out-of-order issue of up to 4 operations per cycle, 64 entry re-order buffer
Mem Disambig	load/store queue, loads may execute when all prior store addresses are known
Registers	32 integer, 32 floating point
Func Units	2-integer ALU, 2-load/store units, 1-FP adder, 1-integer MULT/DIV, 1-FP MULT/DIV
Virtual Mem	8K byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete

Table 1: Baseline Simulation Model.

3 Methodology

To perform our study, we collected information for ten SPEC 2000 programs `applu`, `apsi`, `art`, `bzip`, `facerec`, `galgel`, `gcc`, `gzip`, `mcf`, and `vpr` all with reference inputs. All programs were executed from start to completion using SimpleScalar [5] and Wattch [4]. Because of the lengthy simulation time incurred by executing all of the programs to completion, we chose to focus on only 10 programs. We chose the above 10 programs since their phase based behavior represents a reasonable snapshot of the SPEC 2000 benchmark suite, along with picking some of the programs that showed the most interesting phase-based behavior. Each program was compiled on a DEC Alpha AXP-21164 processor using the DEC C, and FORTRAN compilers. The programs were built under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`).

The timing simulator used was derived from the SimpleScalar 3.0 tool set [5], a suite of functional and timing simulation tools for the Alpha AXP ISA. The baseline microarchitecture model is detailed in Table 1. In addition to this, we wanted to examine energy usage optimizations, so we used a version of Wattch [4] to capture this information. We modified all of these tools to log and reset the statistics every 10 million instructions, and we use this as a base for evaluation.

4 Phase Capture

In this section we motivate the occurrence of phase-based behavior, describe our architecture for capturing it, and examine the accuracy of using the program behavior in our phase-tracking architecture to identify phase changes for various hardware metrics.

4.1 Phase-Based Behavior

The goal of this research is to design an efficient and general purpose technique for capturing and predicting the run-time phase behavior of programs for the purpose of guiding any optimization seeking to exploit large scale program behavior. Figure 1 helps to motivate our approach to the problem. This figure shows the behavior of two programs, `gcc` and `gzip`, as measured by various different statistics over the course of their execution from start to finish. Each point on the graph is taken over 10 million instructions worth of execution. The metrics shown are the

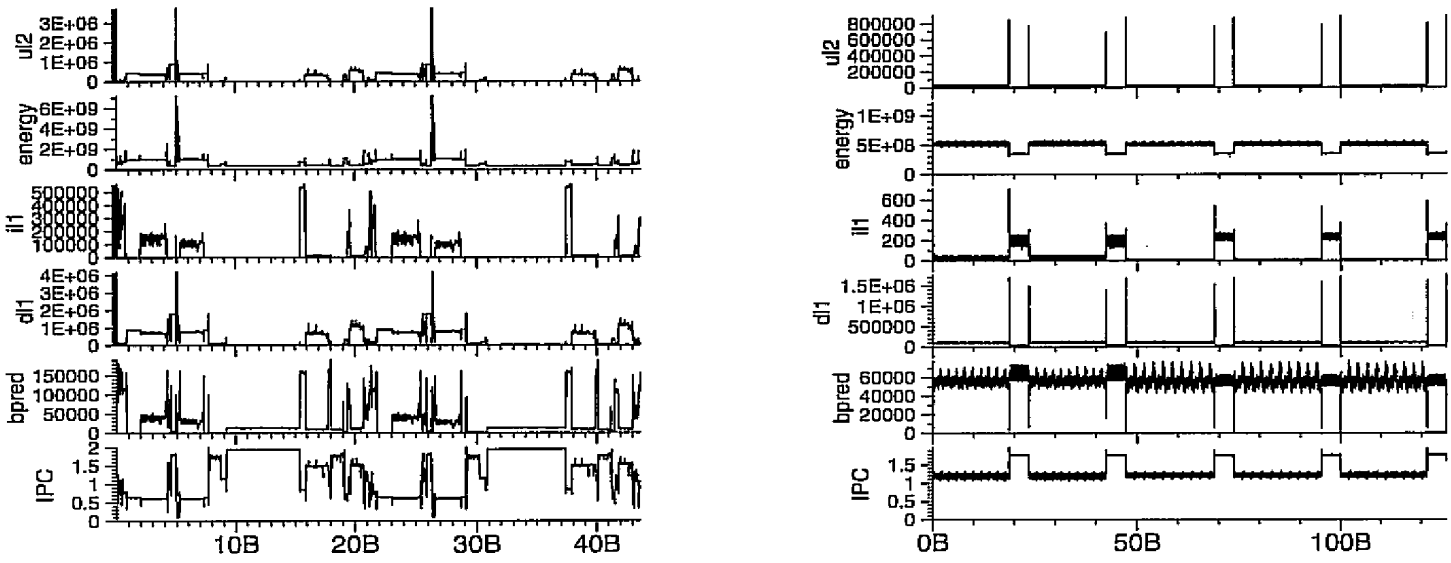

Figure 1: To illustrate the point that phase changes happen across many metrics all at the same time, we have plotted the value of these metrics over billions of instructions executed for the programs `gcc` *(shown left) and* `gzip` *(shown right). Each point on the graph is an average over 10 million instructions. The number of unified L2 cache misses (ul2), the energy consumed by the execution of the instructions, the number of instruction cache (il1) misses, the number of data cache misses (dl1), the number of branch mispredictions (bpred) and the average IPC are plotted.*

number of unified L2 cache misses (ul2), the energy consumed by the execution of the instructions, the number of instruction cache (il1) misses, the number of data cache misses (dl1), the number of branch mispredictions (bpred) and the average IPC. The results show that all of the metrics tend to change in unison, although not necessarily in the same direction. In addition to this, patterns of recurring behavior can be seen over very large time scales.

As can be seen from these graphs, even at a granularity of 10 million instructions (which is at the same time scale as operating system time slices) there can be wildly different behavior seen between intervals. In this paper, we concentrate on a granularity of 10 million instructions because it is both outside the scope of normal architectural timing and is small enough to allow for many complex phase behaviors to be seen.

4.2 Tracking Phases by Executed Code

Our phase tracker architecture operates at two different time scales. It gathers profile information very quickly in order to keep up with processor speeds, while at the same time it compares any data it gathers with information collected over the long term. Additionally, it must be able to do all that while still being reasonable in size.

Our phase profile generation architecture can be seen in Figure 2. The key idea is to capture basic block information during execution, while not relying on any compiler support. Larger basic blocks need to be weighed more heavily as they account for a more significant portion of the execution. To *approximate* gathering basic block information, we capture branch PCs and the number of instructions executed between branches. The input to the architecture is a tuple of information: a branch identifier (PC) and the number of instructions since the last branch PC was executed. This allows us to roughly capture each basic block executed along with the weight of the basic block in terms of the number of instructions executed, as we did in [20, 21] for identifying simulation points.

Classifying phases by examining only the code that is executed allows our phase tracker to be independent of any individual architecture metric. This allows our phase tracker to be used as a general profiling technique building up a profile or database of architecture information on a per phase basis to be used later for hardware or software optimization. Independence from architecture metrics is also very important to allow us to consistently track phase information as the program's behavior changes due to phase-based optimizations.

At this point it is worth making more explicit the differences between our technique and that of Dhodapkar and Smith [10, 9]. Dhodapkar and Smith use a bit vector to track the working set of the code for a particular interval. While our technique is based on the basic block vectors used in [20]. The bit vectors of Dhodapkar and Smith track a metric that is related to which code blocks were *touched*, whereas our metric tracks the *proportion* of time spent executing in each code block. This is a subtle but important distinction. We have found that in complex programs (such as `gcc` and `gzip`) there are many instructions blocks that execute only intermittently. When tracking the pure working set, these infrequently executed blocks can disguise the frequently executed blocks that dominate the behavior of the application. On the other hand, by tracking the frequency of code execution it is possible to distinguish important instructions (basic blocks) from a sea of infrequently executed ones. Examining these differences in more detail is a topic of future research.

Another advantage of tracking the proportions in which the basic blocks are executed is that we can use this information to

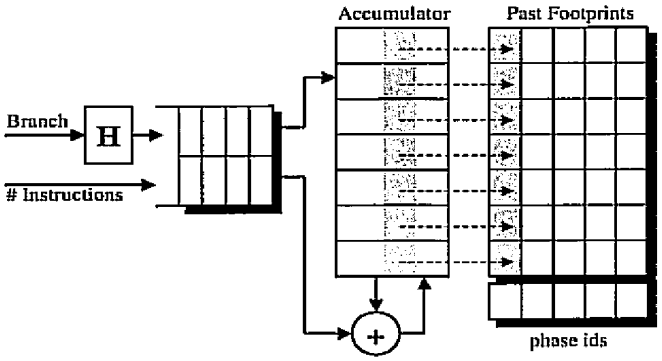

Figure 2: Our phase classification architecture. Each branch PC is captured along with the number of instructions from the last branch. The bucket entry corresponding to a hash of the branch PC is incremented by the number of instructions. After each profiling interval has completed, this information is classified, and if it is found to be unique enough, stored in the past footprint table along with its phase ID.

identify not only when different sections of code are executing, but also when those sections of code are being exercised differently. A simple example is in a graphics manipulation program running a parameterized filter on an input image. If you run a simple 3x3 blur filter on an image you get very different behavior than if you run a 7x7 filter on the same image despite the fact that the same filter code is executing. The 7x7 filter will have many more memory references and those memory references conflict very differently in the cache than in the 3x3 case. We have seen this very behavior in examining the interactive graphics program xv. Using the proportion of execution for each basic block can distinguish these differences, because in the 3x3 filter the head of the loop is called more than twice as frequently as in the 7x7 filter.

The same general idea applies to other data structures as well. Take for example a linked list. As the number of nodes in the linked list traversal changes over different loop invocations, the number of instructions executed inside the loop versus the time spent outside the loop also changes. This behavior can be captured when including a measure of the proportion of the code executed, and this can distinguish between link list traversals of different lengths.

4.3 Capturing the Code Profile

To index into the accumulator table in Figure 2, the branch PC is reduced to a number from 1 to $Nbuckets$ using a hash function. We have found that 32 buckets is sufficient to distinguish between different phases even for some of the more complex programs such as `gcc`. A counter is kept for each bucket, and the counter is incremented by the number of instructions from the last branch to the current branch being processed. Each accumulator table entry is a large (in this study 24-bit), saturating counter, which will not saturate during our profiling interval of 10 million instructions. Updating the accumulator table is the only operation that needs to be performed at a rate equivalent to the processor's execution of the program (once for every branch executed). In comparison, the phase classification described below needs to only be performed once every 10 million instructions (at the end of each interval), and thus is not nearly as performance critical.

We note that the hashing function we use is fundamentally the same as the random projection method we used to generate phases in [21]. In this prior work, we make use of random projections of the data to reduce the dimensionality of the samples being taken. A random projection takes trace data in the form of a matrix of size $L \times B$, where L is the length of the trace and B is the number of unique basic blocks, and multiplies it by a random matrix of size $B \times N$, where N is the desired dimensionality of the data which is much smaller than B. This creates a new matrix of size $L \times N$, which has clustering properties very similar to the original data. The random projection method is a powerful technique when used with clustering algorithms, and for capturing phase behavior as we showed in [21]. The hashing scheme we use in this paper is essentially a degenerate form of random projection that makes a hardware implementation feasible while still having low error. If all the elements of the random projection matrix consist of either a 0 or a 1, and they are placed such that no column of the matrix contains more than a single 1, then the random projection is identical to this simple hashing mechanism. We have designed our phase classification architecture around this principle.

Figure 3 shows the effect of applying the above mentioned technique for capturing the phase behavior of the integer benchmark `gzip`. The x-axis of the figure is in billions of instructions, as is the case in Figure 1. Each point on the y-axis represents an entry of the phase tracker's accumulator table. Each point on the graph corresponds to the value of the corresponding accumulator table entry at the end of a profiling interval. Dark values represent high execution frequency, while light values correspond to low frequency. The same trends that were seen in Figure 1 for `gzip` can be clearly seen in Figure 3. In both of these figures, when observing them at the coarsest granularity, we can see that there are at least three different phases labeled A, B and C. In Figure 3, the phase tracker table entries `2, 5, 7, 13 and 17` distinguish the two identical long running phases labeled A from a group of three long running phases labeled C. Phase table entries `12 and 20` clearly distinguish phase B from both A and C. This figure is pictorial evidence that the phase tracker is able to break the program's execution into the corresponding phases based solely on the executed code, and that these phases correspond to the behavior seen across the different program metrics in Figure 1.

4.4 Forming a Footprint

After the profiling interval has elapsed, and branch block information has been accumulated, the phase must then be classified. To do this we keep a history of past phase information.

If we fix the number of instructions for a profiling interval, then we can divide each bucket by this fixed number to get the percentage of execution that was accounted for by all instructions mapped to that bucket. However, we do not need to know the exact percentages for each bucket. Instead of keeping the

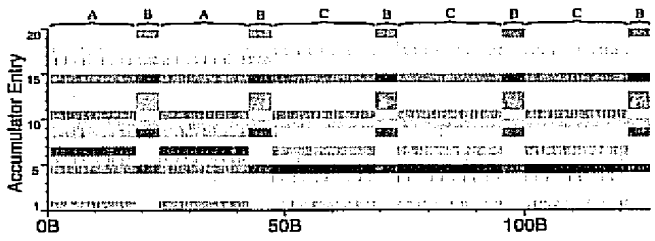

Figure 3: Visualization of the accumulator table used to track program behavior for `gzip`. *The x-axis is in billions of instructions, while the y-axis is the entry of the accumulator table. Each point on the graph corresponds to the value of the accumulator table at the end of a profiling interval where dark values correspond to more heavily accessed entries. The same trends that were seen in Figure 1 can be clearly seen in Figure 3.*

full counter values, we can instead compress phase information down to a couple of the most significant bits. This compressed information will then be kept in the Past Footprint table as shown in Figure 2.

The number of counter value bits that we need to observe is related to $Nbuckets$. As we increase the number of buckets, the data is spread over more buckets (table entries), making for less entries per bucket (better resolution) but at the cost of more area (both in terms of number of buckets and more bits per bucket). To be on the safe side, we would like *any* distribution of data into buckets to provide useful information. To achieve this we need to ensure that even if data is distributed perfectly evenly over all of the buckets, we would still record information about the frequency of those buckets. This can be achieved by reducing the accumulator counter by:

$$(bucket[i] \times Nbuckets)/(intervalsize)$$

If the number of buckets and interval size are powers of two, this is a simple shift operation. For the number of buckets we have chosen (32), and the interval size we profile over, this reduces the bucket size to 6 bits, and thus requires 24 bytes of storage for each unique phase in the Past Footprint table. In practice we see that the top 6 bits of the counter are more than enough to distinguish between two phases. In the worst case, you may need one or two more bits to reduce quantization error, but in reality we have not seen any programs that cause this to be an issue.

If too few buckets are used, aliasing effects can occur due to the hashing function, where two different phases will appear to have very similar Footprints. Therefore, we want to use a sufficiently large number of buckets to uniquely identify the differences in code execution between phases, while at the same time use only a small amount of area.

To examine the aliasing effect and determine what the appropriate number of buckets should be, Figure 4 shows the sum of the differences in the bucket weights found between all sequential intervals of execution. The y-axis shows the sum total of differences for each program. This is calculated by summing the

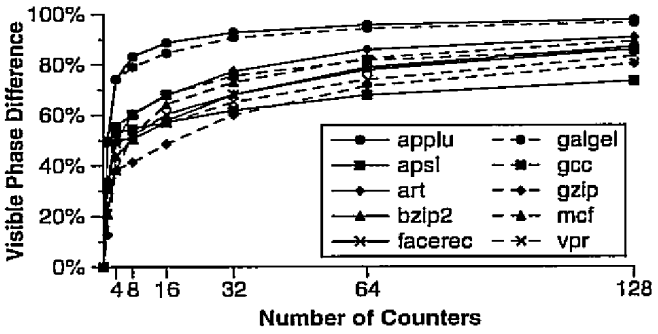

Figure 4: The percent difference found between Footprints from sequential intervals of execution, when varying the number of counters used to represent the footprints. The results are normalized to the difference between intervals found when having an infinite number of buckets to represent the footprint; 32 buckets captures most of the benefit.

differences between the buckets captured for interval i and $i-1$ for each interval i in the program. The x-axis is the number of distinct buckets used. All of the results are compared to the ideal case of using an infinite number of buckets (or one for each separate basic block) to create the Footprint. On the program `gcc` for example, the total sum of differences with 32 buckets was 72% of that captured with an infinite number of buckets. In general we have found that 32 buckets was enough to distinguish between two phases.

4.5 Classifying a Footprint to a Phase ID

After reducing the vector to form a footprint, we begin the classification process by comparing the footprint to a set of representative past footprint vectors. We compare the current vector to each vector in the table. The next section details how we perform the comparison and determine what a match is. If there is a match, we classify the profiled section of execution into the same phase as the past footprint vector, and the current vector is not inserted into the past footprint table. If there is no match, then we have just detected a new phase and hence must create a new unique phase ID into which we may classify it. This is done by choosing a unique phase ID out of a fixed pool of IDs. When allocating a new phase ID, we also allocate a new past footprint entry, set it to the current vector, and store with that entry the newly allocated phase ID. This allows future similar phases to be classified with the same ID. In this way only a single vector is kept for each unique phase ID, to serve as a representative of that phase. After a phase ID is provided for the most recent interval, it is passed along to prediction and statistic logging, and the phase identification part of our algorithm is completed.

To examine the number of phase IDs we need to track, Figure 5 shows the percentage of execution that can be accounted for by the top p phases, where p is shown on the x-axis. Results are graphed for the programs that had the min (`galgel`) and max (`art`) coverage, `gcc`, `gzip`, and the overall average. These results show that most of the program's phase behavior can be captured using a relatively small number of phase IDs.

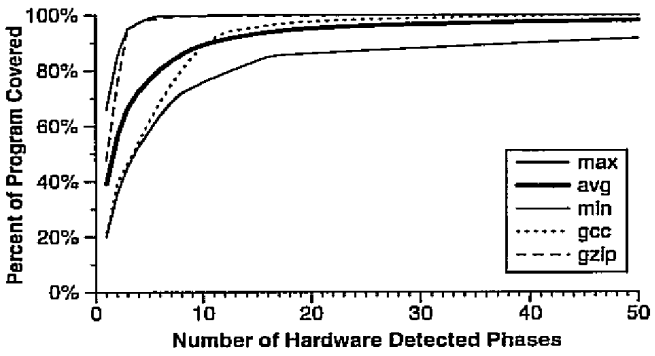

Figure 5: Results of the minimum number of phases that need to be captured versus the amount program execution they cover. The y-axis is the percent of program execution that is covered. The x-axis is the minimum number of phases needed to capture that much program execution.

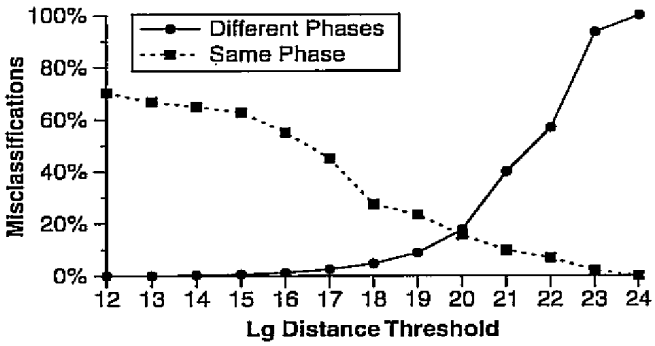

Figure 6: Results showing how well our hardware phase tracker classifies two sequential intervals of execution as being from "Different" or the "Same" phase of execution. The percent of misclassifications are shown in comparison to the phase classifications found using the off-line clustering SimPoint tool [21].

If we only track and optimize for the top 20 phases in each application, we will capture and be able to accurately apply phase prediction/optimizations to over 90% of the program's execution on average. In the worst case (min), we are able to optimize most of the program (over 80%) by only targeting a small number (20) of important recurring phases.

4.5.1 Finding a Match

We search through the Footprint histories to find a match, but this query is complicated by the fact that we are not necessarily searching for an exact match. Two sections of execution that have very similar footprints could easily be considered a match, even if they do not compare exactly. To compare two vectors to one another, we use the Manhattan distance between the two, which is the element-wise sum of the absolute differences. This distance is used to determine if the current interval should be classified as the same phase ID as one of the past footprint intervals.

If we set the distance threshold too low, the phase detection will be overly sensitive, and we will classify the program into many, very tiny phases which will cause us to lose any benefit from doing run-time phase analysis in the first place. If the threshold is too high, the classifier will not be able to distinguish between phases with different behavior. To quantify this effect, we examine how well our hardware technique classifies phases for a variety of thresholds compared to the phases found by the off-line clustering algorithm used in SimPoint [21].

The SimPoint tool is able to make global decisions to optimize the grouping of similar intervals into phases. The off-line algorithm makes no use of thresholds, instead its decisions are based solely on the structure found in the distribution of program behaviors. Our technique must be far more simplistic because it must be performed on-line and with limited computational overhead. This reduction in complexity comes at the cost of increased error.

The Different Phases line in Figure 6 shows the ability of our hardware technique to find phase changes (transitions between one phase and the next) when different thresholds are used to perform the phase classification. For example, when using a Manhattan distance of 1 million as our threshold (shown as 20 on our x-axis because it is in $\log_2$), our hardware technique identified 80% of the phase changes that occurred in the more complex off-line SimPoint analysis. Conversely, 20% of the phase changes were incorrectly classified as having the same phase ID as the last interval of execution.

Likewise, the Same Phases line in Figure 6 represents the ability of our hardware technique to accurately classify two sequential intervals as being part of the same phase as a function of different thresholds (again as compared to the off-line clustering analysis). For example, when using a Manhattan distance of 1 million (shown as 20 on the x-axis), our hardware technique identified 80% of the intervals that stayed in the same phase as correctly staying in the same phase, but 20% of those intervals were classified as having a different phase ID from the prior phase.

A misclassification occurs when two sequential intervals of execution are classified as being in the same phase or in different phases using our hardware approach when the off-line clustering analysis tool found the opposite for these two intervals.

If we are too aggressive and our hardware phase analysis indicates that there are phase changes when there are actually no noticeable changes in behavior, then we will create too many phase IDs that have similar behavior. This can create more overhead for performing phase-based optimization. On the other hand, if we are too passive in distinguishing between different phases, we will be missing opportunities to make phase specific optimizations.

In order to strike a balance between having a high capture rate and reducing the percent of false positives, we chose to use a threshold of 1 million. When comparing this with the interval size of 10 million instructions, this means that a difference in the phase behavior will be detected if 10% of the executed instructions are in different proportions. In choosing 1 million, we have on average a 20% misclassification rate. Note, that a misclassification does not necessarily mean that an incorrect optimization

will be performed. For example, if we have a "Same Phase" misclassification (the two intervals were really from the same phase, but were classified into different phases), then a phase change is observed using our hardware technique when there was not one in the baseline classifier. If the two hardware detected phases have the same optimization applied to them, then this misclassification can have no effect.

4.6 Per-Phase Performance Metric Homogeneity

Using the techniques presented above, we can perform phase classification on programs at run-time with little to no impact on the design of the processor core. One of the goals of phase classification is to divide the program into a set of phases that are fairly homogeneous. This means that an optimization adapted and applied to a single segment of execution from one phase, will apply equally well to the other parts of the phase. In order to quantify the extent to which we have achieved this goal, we need to test the homogeneity of a variety of architectural statistics on a per-phase basis.

Figure 7 shows the results of performing this analysis on the phases determined at run-time. Due to space constraints we only show results for two of the more complicated programs `gcc` and `gzip`. For both programs, a set of statistics for each phase is shown. The first phase that is listed (separated from the rest) as `full`, is the result of classifying the entire program into a single phase. The results show that for `gcc` for example, the average IPC of the entire program was 1.32, while the average number of cache misses was 445,083 per ten million instructions. In addition to just the average value, we also show the standard deviation for that statistic. For example, while the average IPC was 1.32 for `gcc`, it varied with a standard deviation of over 43% from interval to interval. If the phase-tracking hardware is successful in classifying the phases, the standard deviations for the various metrics should be low for a given phase ID.

Underneath the phase marked `full` are the five most frequently executed phases from the program as identified by our phase tracker. The phases are weighted by the percentage of the program's executed instructions they account for. For `gcc`, the largest phase accounts for 18.5% of the instructions in the entire program and has an average IPC of 0.61 and a standard deviation of only 1.6% (of 0.61). The other top four phases have standard deviations at or below this level, which means that our technique was successful at dividing up the execution of `gcc` into large phases with similar execution behavior with respect to IPC. Note, that some metrics for certain phases have a high standard deviation, but this occurs for architecture features/metrics that are unimportant for that phase. For example, the phase that occurs for 7.2% of execution in `gcc` has only 75 L1 instruction cache misses on average. This is an L1 miss rate of 0.00075%, so an error of 215% for this metric will not likely have any effect on the phase.

When we look at the energy consumption of `gcc`, it can be observed that energy consumption swings radically (a standard deviation of 90%) over the complete execution of the program. This can be seen visually in Figure 1, which plots the energy usage versus instructions executed. However, after dividing the program into phases, we see that each phase has very little variation within itself. All have less than 2% standard deviation. By analyzing `gcc` it can also be seen that the phase partitioning does a very good job across all of the measured statistics even though only *one* metric is used. This indicates that the phases that we have chosen are in some way representative of the actual behavior of the program.

5 Phase Prediction

The prior section described our phase tracking architecture, and how it can be used to classify phases. In this section we focus on using phase information to predict the next phase. For a variety of applications it is important to be able to predict future phase changes so that the system can configure for the code it will soon be executing rather than simply reacting to a change in behavior.

Figure 8 shows the percentage interval transitions that are changes in phase, for our set of benchmarks. For all of these programs, phase changes come quite often, but it should be noted that this statistic alone cannot gauge the complexity of the program behavior. The program `gcc` switches less than 10% of the time but switches between *many* different phases. The other extreme is `art` which switches almost half the time, but it is only switching between a few distinct phases. In this case, large repeating patterns can be observed. No two phases executing sequentially are that similar, but there is an order to the sequence. By adding in a prediction scheme for these cases, we not only take advantage of stable conditions as in past research, but actually take advantage of any repeating patterns in program behavior.

5.1 Markov Predictor

The prediction of phase behavior is different from many other systems in which hardware predictors are used. Because of this new environment, a new type of predictor has the potential to perform better than simply using predictors from other areas of computer architecture (branch and address prediction, memory disambiguation, etc.).

After observing the way that phases change, we determined that two pieces of information are important. First, the set of phases leading up to the prediction are very important, and second, the *duration* of execution of those phases is important.

A classic prediction model that is easily implementable in hardware is a Markov Model. Markov Models have been used in computer architecture to predict both prefetch addresses [13] and branches [8] in the past. The basic idea behind a Markov Model is that the next state of the system is related to the last set of states.

The intuition behind this design is that phase information tends to be characterized by many sections of stable behavior interspersed with abrupt phase changes. The key is to be able to predict when these phase changes will occur, and to know ahead of time what phase they will change to. The problem is that the changes are often preceded by stable conditions, and if we only consider the last couple of intervals we will not be able to tell the difference between sections of stable behavior that precede a phase change, and those sections that will continue to be stable. Instead, we need a way of compressing down stable phase

	phase	IPC	(stddev)	bpred	(stddev)	dl1	(stddev)	il1	(stddev)	energy	(stddev)	ul2	(stddev)
gcc	full	1.32	(43.4%)	27741	(135.5%)	445083	(110.7%)	50763	(203.2%)	6.44E+08	(90.0%)	227912	(139.7%)
	18.5%	0.61	(1.6%)	34665	(22.0%)	753382	(5.4%)	125091	(23.2%)	1.03E+09	(1.8%)	395997	(5.3%)
	18.1%	1.95	(0.3%)	13048	(3.9%)	28112	(15.1%)	43	(73.9%)	3.22E+08	(0.2%)	1006	(5.6%)
	7.2%	0.64	(0.2%)	843	(15.1%)	885081	(0.1%)	75	(215.5%)	9.78E+08	(0.3%)	443655	(0.1%)
	4.0%	1.49	(1.2%)	10145	(7.6%)	703554	(6.8%)	15591	(5.2%)	4.20E+08	(1.1%)	354084	(7.0%)
	3.9%	1.76	(1.6%)	2015	(13.6%)	98947	(5.9%)	102	(45.1%)	3.57E+08	(1.6%)	15595	(12.6%)
	phase	IPC	(stddev)	bpred	(stddev)	dl1	(stddev)	il1	(stddev)	energy	(stddev)	ul2	(stddev)
gzip	full	1.33	(16.3%)	56045	(11.1%)	90446	(58.2%)	60	(138.1%)	4.82E+08	(13.5%)	22880	(112.0%)
	17.1%	1.24	(3.4%)	53300	(10.8%)	96960	(10.1%)	12	(44.2%)	5.05E+08	(3.5%)	24218	(8.6%)
	9.4%	1.23	(3.8%)	54973	(11.5%)	99523	(11.3%)	11	(45.5%)	5.09E+08	(3.8%)	24518	(9.3%)
	8.8%	1.76	(0.6%)	56449	(4.8%)	37331	(5.6%)	241	(8.4%)	3.55E+08	(0.6%)	5617	(15.6%)
	8.0%	1.22	(4.3%)	54791	(6.8%)	99671	(11.9%)	40	(25.7%)	5.14E+08	(4.4%)	28153	(11.0%)
	7.4%	1.24	(3.1%)	55215	(11.1%)	96701	(9.6%)	12	(35.4%)	5.04E+08	(3.2%)	23701	(8.4%)

Figure 7: Examination of per-phase homogeneity compared to the program as a whole (denoted by `full`). For the two programs and each of the top 5 phases of each program, we show the average value of each metric and the standard deviation. The name of the phase is the percent of execution that it accounts for in terms of instructions. These results show that after dividing up the program into phases using our run-time scheme the behavior within each phase is quite consistent.

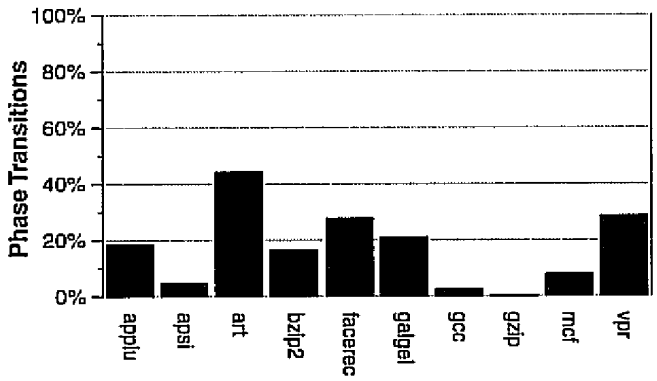

Figure 8: The percent of execution intervals that transition to a different phase from the prior execution interval's phase as found by our phase tracking architecture with 32 footprint counters using a 1 million Manhattan threshold.

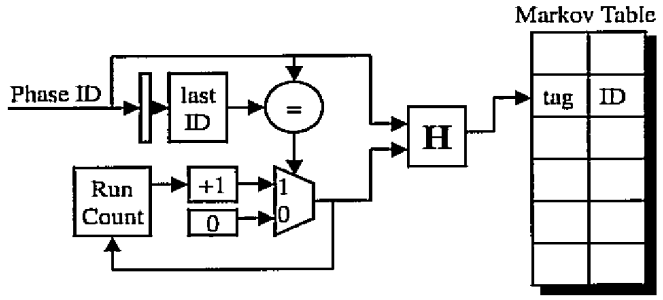

Figure 9: Phase Prediction Architecture for the Run Length Encoded (RLE) Markov predictor. The basic idea behind the predictor is that two pieces of information are used to generate the prediction, the phase id that was just seen, and the number of times prior to now that it has been seen in a row. The index into the prediction table is a hash of these two numbers.

information into a piece of information that we can use as state.

5.2 Run Length Encoding Markov Predictor

To compress the stable state we use a *Run Length Encoding* (RLE) Markov predictor. The basic idea behind the predictor is that it uses a run-length encoded version of the history to index into a prediction table. The index into the prediction table is a hash of the phase identifier and the number of times the phase identifier has occurred in a row.

Figure 9 shows our RLE Markov Phase ID prediction architecture. The the lower order bits of the hash function provide an index into the prediction table, and the higher order bits of the hash function provide a tag. When there is a tag match, the phase ID stored in the table provides a prediction as to the next phase to occur in execution. When there is a tag miss, the prior phase ID is assume to be the next phase ID to occur in the program's execution. We found that predicting the last phase ID to be 75% accurate on average.

We only update the predictor when there is (1) a change in the phase ID, or (2) when there is a tag match. We only insert an entry when there is a phase ID change, since we want to predict when the phase is going to change. Execution intervals where the same phase ID occurs several times in a row do not need to be stored in the table, since they will be correctly predicted as "last phase ID", when the there is a table miss. This helps table capacity constraints and avoids polluting the table with last phase predictions. For the second update case, when there is a tag match, we update the predictor because the observed run length may have potentially changed.

5.3 Predictor Comparison

We compare our RLE Markov phase predictor with other prediction schemes in Figure 10. This Figure has four bars for every program, and each bar corresponds to the prediction accuracy of a prediction architecture. The first and simplest scheme, `Last Phase`, simply predicts that the next phase is the same as the current phase, in essence always predicting stable operation. The prediction accuracy of this scheme is inversely proportional to the rate at which phases change in a given benchmark. For the program `gzip` for example, there are long periods of execution where the phase does not change, and therefore predicting no-change does exceptionally well.

In order to insure that we were not simply providing an

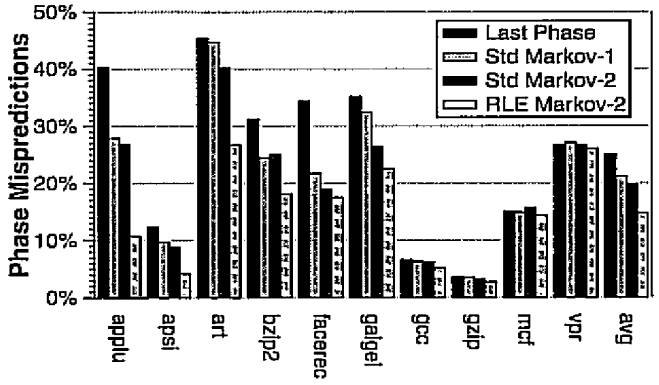

Figure 10: Phase ID Prediction Accuracy. This figure shows how well different prediction schemes work. The most naive scheme, `last`*, simply predicts that the phases never change. The bars marked* `Markov` *and* `RLE Markov` *show how well we can predict the phase identifiers if we use a Markov prediction scheme with a Markov table size of 256 entries.*

expensive filter for noise in the phase IDs, we also compared against a simple noise filter which works by predicting that the next phase will be the most commonly occurring of the last three phases seen. This is not shown, as it actually performed worse on all of the programs.

Additionally we wanted to examine the effect of a simple Markov model predictor for history lengths of 1 and 2. The Markov model predictor does a better job of predicting phase transitions than `Last Phase`, but it is limited by the fact that long runs will always be predicted as infinitely stable due to the history filling up. However, it is still very effective for `facerec` and `applu`, but does not provide much benefit for either `art` or `galgel`.

The final bar, `RLE Markov`, is our improved Markov predictor which compresses stable phases into a tuple of phase id and duration. All of the Markov predictors simulated had 256 entries taking up less than 500 bytes of storage. Using RLE Markov outperforms both the `Last Phase` and traditional `Markov` on all the benchmarks. It performs especially well compared to other schemes on both `applu` and `art`. Overall, using a Run-Length Encoded Markov predictor can cut the phase mispredictions down to 14% on average.

6 Applications

This section examines three optimization areas in which a phase-aware architecture can provide an advantage. We begin by examining the relationship between phase behavior and value locality. We then demonstrate ways to reduce processor energy consumption by adjusting the aggressiveness of the data cache and the instruction front end.

6.1 Frequent Value Locality

Prior work on value predictors has shown that there is a great deal of value locality in a variety of programs [14, 7]. Recently, researchers have started to take advantage of frequently loaded values for the purpose of optimizing caches. For example, Yang and Gupta [22] proposed a data cache organization that compresses the most frequently used program values in order to save energy. Another way of exploiting value locality is through value specialization, which can be done either statically or dynamically [6, 17, 16] to create specialized versions of procedures or code-regions based upon the values frequently seen. These techniques are built on the idea of finding the most frequent values for loads over the whole program, and then specializing the program to those frequent values.

We examine the potential of capturing frequent values on a per-phase basis and compare this to the frequent values aggregated over the entire program, as would be used in value code specialization [6]. To perform this experiment we first gathered the top 16 values that were loaded over the complete execution of the program and stored them into a table. We then examined the percentage of executed loads that found their loaded value in this table. This result is shown as `Static` in Figure 11. While significant portions of some programs are covered by just these few top values (such as `applu`), over half of the programs have less than 10% of their loaded values covered by these top values.

The question is: can we do better by exploiting hardware-detected phase information? To answer this question we take the top 16 values for each phase, as detected by the hardware phase tracker. These top values will be shared across a single phase even if it is split into two or more different sections of execution. Each load in the program is then checked against the top values for its corresponding phase. The `Phase Coverage` bar in Figure 11 shows the percent of all load values in the program that were successfully matched to it's per-phase top value set.

Without any notion of loads or values, our method of dividing up phases is very successful at assisting in the search for frequent values. By just tracking the top 16 values of each phase, we are able to capture the values from almost 50% of the executed loads on average. The `Perfect` bar shows percentage of loads covered if one captures the top 16 load values for each and every interval (i.e., 10 million instructions) separately. This is in effect the best that we could hope to achieve for an interval size of 10 million instructions, because the 16 entries in the value table are custom crafted for each interval individually. As shown in Figure 11, the phase-tracker compares favorably with the optimal coverage. Two thirds of the total possible benefit from per-interval value locality can be captured by per-phase value locality. It is important to point out this graph by itself is not a good indicator of usefulness as near perfect coverage could be achieved simply by making every interval a separate phase. However, as shown in Figure 5 only a few phases (around 20) are used to cover at least 80% of the program's execution.

6.2 Dynamic Data Cache Size Adaptation

In a modern processor a significant amount of energy is consumed by the data cache, but this energy may not be put to good use if an application is not accessing large amounts of data with high locality. To address this potential inefficiency, previous work has examined the potential of dynamically reconfiguring the data caches with the intention of saving power. In [2], Balasubramonian et. al. present two different schemes with

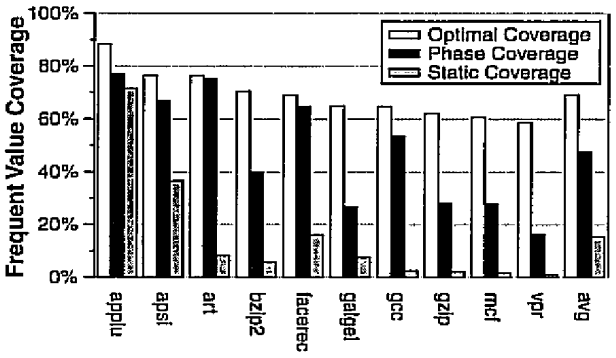

Figure 11: The percent of the program's load values that are found in a table of the most frequently values loaded over the whole program (Static Coverage), on a per-phase basis (Phase Coverage), and on a per execution interval basis (Optimal Coverage).

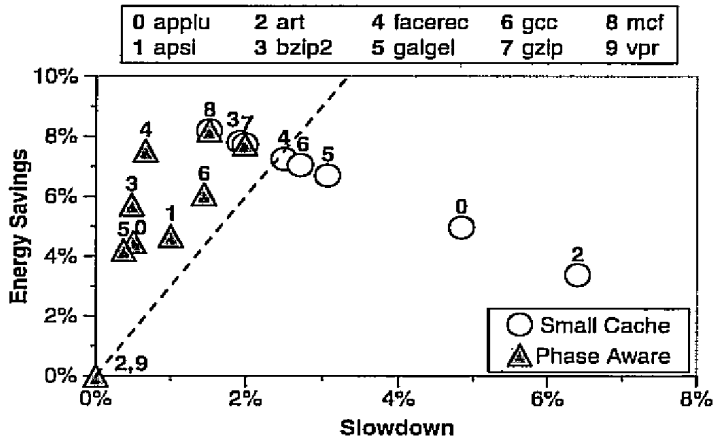

Figure 12: Data Cache Re-configuration. The tradeoff between energy savings and slowdown for two different cache policies. All results are relative to a 32KB 4-way associative cache. The circles in the graph (each labeled with a number for the program the data point is from) show the energy and performance of an 8KB direct mapped cache. The triangles show the tradeoff of intelligently switching between an 8KB direct mapped and a 32KB 4-way data cache based on phase classification and prediction.

which re-configuration may be guided. In one scheme, hardware performance counters are read by re-configuration software every hundred thousand cycles. The software then makes a decision based on the values of the counters. In another scheme, re-configuration decisions are performed on procedure boundaries instead of at fixed intervals. To reduce the overhead of re-configuration, software to trigger re-configuration is only placed before procedures that account for more than a certain percentage of execution.

Another form of re-configurable cache that has been proposed dynamically divides the data cache into multiple partitions, each of which can be used for a different function such as instruction reuse buffers, value predictors, etc [18]. These techniques can be triggered at different points in program execution including procedure boundaries and fixed intervals. The overhead of re-configuration can be quite large and making these policy decisions only when the large scale program behavior changes, as indicated by phase shifts in our hardware tracker, can minimize overhead while guaranteeing adequate sensitivity to attain maximum benefit.

We examined the use of phase tracking hardware to guide an energy aware, re-sizable cache. The energy consumption of the data cache can be reduced by dynamically shifting to a smaller, less associative cache configuration for program phases that do not benefit significantly from more aggressive cache configurations. By targeting only those phases that are predicted to have energy savings due to cache size reduction, our scheme is able to reduce power with very little impact on the performance.

We examined an architecture with two possible cache configurations, 32KB 4-way associative and 8KB direct mapped. In Figure 12, the trade off between these two configurations is plotted. For each program, we use the 32KB cache configuration as the baseline result. The labeled circles in Figure 12 show the total processor energy savings and performance degradation for each program if only the smaller (8KB) cache size is used. For example, a processor with a smaller cache configuration for the program `applu` is both 5% slower and uses 5% less energy. Two programs, `vpr` and `apsi`, actually use more energy with a smaller cache due to large slow downs. These two points are off the scale of this graph and are not shown.

While examining energy savings and slow down is interesting, it is important to note that there is more than one way to reduce both energy and performance. Voltage scaling in particular has proven to be a technology capable of reaping large energy savings for a relative reduction in performance. For our results, we assume that for voltage scaling a performance degradation of 5% will yield an approximate energy saving of 15%. We use this rule of thumb as our guideline for determining when to reduce the active size of the cache. In Figure 12, this simple model of voltage scaling is plotted as a dashed line. When the cache size is reduced, most programs fall far short of this baseline, meaning that voltage scaling would provide a better performance-energy tradeoff. There are a couple of exceptions, in particular `mcf`, `bzip`, and `gzip` do well even without any sort of phase-based re-configuration.

The shaded triangles in Figure 12 show what happens if we use phase classification and prediction to guide our re-configuration. When a new phase ID is seen, we sample the IPC and energy used for a few intervals using the 32KB 4-way cache, and a few intervals for the 8KB direct mapped cache. These samples could be kept in a small hardware profiling table associated with the phase ID. After taking these samples, if we find that a particular phase is able to achieve more than three times the energy savings relative to the slow down seen when using the 8KB cache, we then predict for this phase ID that the smaller cache size should be used. This heuristic means that the small cache size is used only if re-configuration would beat voltage scaling for that phase. After a decision has been made as to the con-

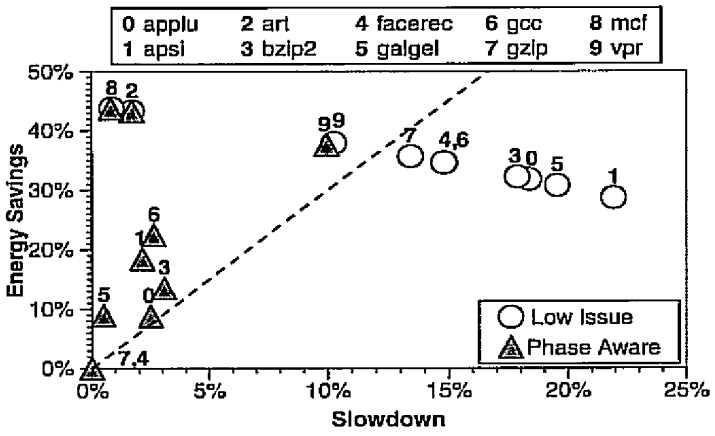

Figure 13: Processor Width Adaptation. The tradeoff between energy savings and slowdown for two different front end policies. All results are relative to an aggressive 8–issue machine. The circles in the graph (each labeled with a number for the program) show the energy and performance of a less aggressive 2–issue processor. The triangles show using the phase classifier and predictor for switching between 2–issue and 8–issue based on phase changes.

figuration to use for a phase ID, the corresponding cache size is stored in the phase profiling table/database associated with that phase ID. The phase classifier and predictor are then used to predict when a phase change occurs. When a phase change prediction occurs, the predicted phase ID looks up the cache size in the profiling table, and re-configures the cache (if it is not already that size) at the predicted phase change.

For all programs, our re-configuration is able to beat or tie voltage scaling. For example, using phase-based reconfiguration results in a slowdown of 0.5% for `applu`, while the total energy savings is 4.5%. Even the program `apsi`, which had *increased* energy consumption in the small cache configuration, is able to get almost 5% energy savings with only a 1% slowdown.

6.3 Dynamic Processor Width Adaptation

One way to reduce the energy consumption in a processor is to reduce the number of instructions entering the pipeline every cycle [12, 1]. We call this adjusting the width of the processor. Reducing the width of the processor reduces the demand on the fetch, decode, functional units, and issue logic. Certain phases can have a high degree of instruction level parallelism, whereas other phases have a very low degree. Take for example the top two phases for `gcc` shown in Figure 7. The intervals classified to be in the first phase consisting of 18.5% of execution have an IPC of 0.61 with a high data cache miss rate. In comparison, the intervals in the second most frequently encountered phase (accounting for 18.1% of execution) have an IPC of 1.95 and very low data cache miss rates. We can potentially save energy without hurting performance by throttling back the width of the processor for phases that have low IPC, while still using aggressive widths for phases with high IPC.

In the current literature, decisions to reduce or increase the fetch/decode/issue bandwidth of the processor are made either at fixed intervals (relatively short intervals such as 1,000 cycles) [12] or, as in the case of branch confidence based schemes, when a branch instruction is fetched [1]. It can very difficult to design real systems that save energy by reconfiguring at these speeds, but a hardware phase-tracker can help make these decisions at a coarser granularity while still maintaining performance and energy benefits.

We examined an architecture that could be configured with 2 different widths - one where up to 2 instructions are decoded and up to 2 issued per cycle, and one where up to 8 instructions are decoded and up to 8 issued per cycle. When a new phase ID is seen by the phase tracker, we sample the IPC for three intervals with a width of 2 instructions, and three intervals with a width of 8 instructions. If there is little difference in the IPC between these two widths, then we assign a width of 2 instructions to this Phase ID in our profiling table, otherwise we assign a width of 8 instructions. During execution, we use the phase ID predictor to effectively predict the width for the next interval of execution and adjust the processor's width accordingly. Our results show that the chosen configuration for a given phase can be trained (1) with only a few samples, and (2) only once to accurately represent the behavior of a given phase ID. This requires very little training time due to the fact that 20 or fewer phase IDs are needed to capture 80% or more of a program's execution as shown in Figure 5.

Figure 13 is a graph of the results seen when applying phase-directed width re-configuration. The white circles in the graph show the behavior of running the programs on only a 2–wide machine relative to the more aggressive 8–wide machine. The dotted line again shows what could potentially be achieved if voltage scaling was used. While `mcf` and `art` save a lot of energy with little performance degradation on a 2–wide machine, the other programs do not fair as well. The program `apsi`, for example, has a slowdown of over 22% with an energy savings of around 30%. This does not compare favorably to voltage scaling (as discussed in Section 6.2). On the other hand if we use phase-directed width throttling on `apsi`, a total processor energy savings of 18% can be achieved with only 2.2% slowdown.

For all of the programs we examined, with one exception, the slowdown due to phase aware width throttling was less than 4%, while the average energy savings was 19.6%. This result demonstrates that there is significant benefit to be had in the reconfiguration of processor front end resources even at very large granularities. In the worst case, this will mean a re-configuration every 10 million instructions, and on average every 70 million instructions. This should be designable even under conservative assumptions.

7 Summary

In this paper we present an efficient run-time phase tracking architecture that is based on detecting changes in the code being executed. This is accomplished by dividing up all instructions seen into a set of buckets based on branch PCs. This way we approximate the effect of taking a random projection of the basic

block vector, which was shown in [21] to be an effective method of identifying phases in programs.

Using our phase classification architecture with less than 500 bytes of on-chip memory, we show that for most programs, a significant amount of the program (over 80%) is covered by 20 or less distinct phases. Furthermore, we show that these phases, while being distinct from one another, have fairly uniform behavior within a phase, meaning that most optimizations applied to one phase will work well on all intervals in that phase. In the program `gcc`, the IPC attained by the processor on average over the full run of execution is 1.32, but has a standard deviation of more than 43%. By dividing it up into different phases, we achieve much more stable behavior, with IPCs ranging between 0.61 and 1.95, but now with standard deviations of less than 2%.

In addition to this, we present a novel phase prediction architecture using a Run Length Encoding Markov predictor that can predict not only when a phase change is about to occur, but to which phase ID it will transition to. In using this design, which also uses less than 500 bytes of storage, we achieve a phase prediction miss rate of 10% for `applu` and 4% for `apsi`. In comparison, always predicting that the phase will stay the same results in a miss rate of 40% and 12% respectively.

We also examined using our phase tracking and prediction architecture to enable new phase-directed optimizations. Traditional architecture and software optimizations are targeted at the average or aggregate behavior of a program. In comparison, phase-directed optimizations aim at optimizing a program's performance tailored to the different phases in a program. In this paper, we examined using phase tracking and prediction to increase frequent value profiling coverage, and to provide energy savings through data cache and processor width re-configuration.

We believe our phase tracking and prediction design will open the door for a new class of run-time optimization that targets large scale program behavior. Even though we present a hardware implementation for phase tracking, a similar design can be implemented in software to perform phase classification for run-time optimizers, just-in-time compilation systems, and operating systems. Hardware and software optimizations that can potentially benefit the most from phase classification and prediction are (1) those that need expensive profiling/training before applying an optimization, (2) those where the time or cost it takes to perform the optimization is either slow or expensive, and (3) those that can benefit from specialization where they have the same code/data being used differently in different phases of execution. By using our dynamic phase tracking and prediction design, phase-behavior can be characterized and predicted at the largest of scales, providing a unified mechanism for phase-directed optimization.

Acknowledgments

We would like to thank Jeremy Lau and the anonymous reviewers for providing useful comments on this paper. This work was funded in part by NSF CAREER grant No. CCR-9733278, Semiconductor Research Corporation grant No. SRC-2001-HJ-897, and an equipment grant from Intel.

References

[1] J.L. Aragon, J. Gonzalez, and A. Gonzalez. Power-aware control speculation through selective throttling. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture*, February 2003.

[2] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *33rd International Symposium on Microarchitecture*, pages 245–257, 2000.

[3] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W. W. Hwu. Vacuum packing: Extracting hardware-detected program phases for post-link optimization. In *35th International Symposium on Microarchitecture*, December 2002.

[4] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.

[5] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, U. of Wisconsin, Madison, June 1997.

[6] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, March 1999.

[7] B. Calder, G. Reinman, and D.M. Tullsen. Selective value prediction. In *26th Annual International Symposium on Computer Architecture*, pages 64–74, June 1999.

[8] I.-C. Chen, J. T. Coffey, and T. N. Mudge. Analysis of branch prediction via data compression. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 128–137, October 1996.

[9] A. Dhodapkar and J. E. Smith. Dynamic microarchitecture adaptation via co-designed virtual machines. In *International Solid State Circuits Conference*, February 2002.

[10] A. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *29th Annual International Symposium on Computer Architecture*, May 2002.

[11] M. Huang, J. Renau, and J. Torrellas. Profile-based energy reduction in high-performance processors. In *4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2001.

[12] A. Iyer and D. Marculescu. Power aware microarchitecture resource scaling. In *Proceedings of the DATE 2001 on Design, automation and test in Europe*, pages 190–196, 2001.

[13] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[14] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. Value locality and load value prediction. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, October 1996.

[15] M. Merten, A. Trick, R. Barnes, E. Nystrom, C. George, J. Gyllenhaal, and Wen mei W. Hwu. An architectural framework for run-time optimization. *IEEE Transactions on Computers*, 50(6):567–589, June 2001.

[16] M. Mock, C. Chambers, and S.J. Eggers. Calpa: a tool for automating selective dynamic compilation. In *33rd International Symposium on Microarchitecture*, pages 291–302, December 2000.

[17] R. Muth, S.A. Watterson, and S.K. Debray. Code specialization based on value profiles. In *Static Analysis Symposium*, pages 340–359, 2000.

[18] P. Ranganathan, S. V. Adve, and N.P. Jouppi. Reconfigurable caches and their application to media processing. In *27th Annual International Symposium on Computer Architecture*, pages 214–224, June 2000.

[19] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report UCSD-CS99-630, UC San Diego, August 1999.

[20] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[21] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[22] J. Yang and R. Gupta. Frequent value locality and its applications. *Special Issue on Memory Systems, ACM Transactions on Embedded Computing Systems*, 1(1):79–105, November 2002.

EXHIBIT 9
TO DECLARATION OF
JAMES A FLIGHT

[Search]

Reports Listed by Year

- 1999
  - On the Resilience of Broadcasting Strategies in a Failure-Propogating Environment . CS1999-0610, Meng-Jang Lin, Aleta M Ricciardi and Keith Marzullo; July 6, 1999
  - Selecting tile shape for minimal execution time. CS1999-0616, Karin Hogstedt, Larry Carter and Jeanne Ferrante; May 20, 1999
  - Determining the idle time of a tiling. CS1999-0617, Karin Hogstedt, Larry Carter and Jeanne Ferrante; July 6, 1999
  - Adaptive Performance Prediction for Distributed Data-Intensive Applications. CS1999-0619, Marcio Faerman, Alan Su, Richard Wolski and Francine Berman; May 18, 1999
  - Directional Gossip: Gossip in a Wide Area Network. CS1999-0622, Meng-Jang Lin and Keith Marzullo; June 16, 1999
  - A Client-Server Oriented Algorithm for Virtually Synchronous Group Membership in WANs. CS1999-0623, Idit Keidar, Jeremy Sussman, Keith Marzullo and Danny Dolev; July 7, 1999
  - Minimax Programs and Bitonic Column Matrices. CS1999-0624, Paul A. Tucker and T. C. Hu; June 17, 1999
  - Min Cuts Without Path Packing. CS1999-0625, Paul A. Tucker, T. C. Hu and M. T. Shing; June 17, 1999
  - PCA = Gabor for Expression Recognition. CS1999-0629, Matthew N. Dailey and Garrison W. Cottrell; October 26, 1999
  - Application Scheduling over Supercomputers: A Proposal. CS1999-0631, Walfredo Cirne and Fran Berman; October 7, 1999
  - Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. CS1999-0632, Henri Casanova, Arnaud Legrand, Dmitrii Zagorodnov and Fran Berman; October 14, 1999
  - Proofs on Safety for Untrusted Code. CS1999-0633, Grigore Rosu and Nathan Segerlind; October 27, 1999
  - Optimistic Virtual Synchrony. CS1999-0634, Jeremy Sussman, Idit Keidar and Keith Marzullo; November 9, 1999
  - Gossip versus Deterministic Flooding: Low Message Overhead and High Reliability for Broadcasting on Small Networks. CS1999-0637, Meng-Jang Lin, Keith Marzullo and Stefano Masini; November 18, 1999
  - Agent Usage Patterns: Bridging the Gap Between Agent-Based Applications and Middleware. CS1999-0638, Eugene Hung and Joseph Pasquale; November 19, 1999
  - AspectBrowser: Tool Support for Managing Dispersed Aspects. CS1999-0640, W. G. Griswold, Y. Kato and J. J. Yuan; January 3, 2000
- 2000
  - Limited Mobile Agents: A Practical Approach. CS2000-0641, Jesse M. Steinberg and Joseph Pasquale; December 29, 1999
  - Combining Workstations and Supercomputers to Support Grid Applications: The Parallel Tomography Experience. CS2000-0642, Shava Smallen, Walfredo Cirne, Jaime Frey, Fran Berman, Rich Wolski, Mei-Hui Su, Carl Kesselman, Steve Young and Mark Ellisman; January 7, 2000
  - Application Scheduling on the Information Power Grid. CS2000-0644, Dmitrii Zagorodnov, Francine Berman and Rich Wolski; January 11, 2000
  - Encode-then-encipher encryption: How to exploit nonces or redundancy in plaintexts for efficient cryptography. CS2000-0646, Mihir Bellare and Phillip Rogaway; March 6, 2000
  - Circular Coinduction. CS2000-0647, Grigore Rosu and Joseph Goguen; March 14, 2000
  - Reducing the Overhead of Compilation Delay. CS2000-0648, Chandra Krintz, David Grove, Derek Lieber, Vivek Sarkar and Brad Calder; March 27, 2000
  - Reducing DRAM Power Using Compiler Assisted Refreshing. CS2000-0649, Timothy Sherwood and Brad Calder; April 21, 2000
  - Dynamic Selection of Compression Formats to Reduce Transfer Delay. CS2000-0650, Chandra Krintz and Brad Calder; April 21, 2000
  - Scalable Causal Message Logging for Wide-Area Networks. CS2000-0651, Karan Bhatia, Keith Marzullo and Lorenzo Alvisi; April 21, 2000
  - Abstract Semantics for Module Composition. CS2000-0653, Grigore Rosu; May 8, 2000
  - Plane Cover Multiple Access: A New Approach to Maximizing Cellular System Capacity. CS2000-0654, Paul Blair and George Polyzos; May 28, 2000
  - Multi-Language Support in a Program Analysis and Visualization Tool. CS2000-0655, Stuart Moskovics; June 20, 2000
  - Design Automation for Finite State Machine Predictors. CS2000-0656, Timothy Sherwood and Brad Calder; June 28, 2000
  - A Power Efficient Speculative Fetch Architectur. CS2000-0657, Glenn Reinman, Brad Calder and Todd Austin; June 28, 2000
  - Uniform Hashing with Multiple Passbits. CS2000-0658, Paul Martini and Walter Burkhard; August 18, 2000
  - Teaching Software Engineering in a Compiler Project Course. CS2000-0659, William G. Griswold; September 12, 2000
  - Exploiting the Map Metaphor in a Tool for Software Evolution. CS2000-0660, William G. Griswold, Jimmy J. Yuan and Yoshikiyo Kato; September 20, 2000
  - Hurwitz Interconnect Delay Evaluation - HIDE: User's Manual. CS2000-0661, Xiao-Dong Yang, Zhanhai Qin and Chung-Kuan Cheng; November 9, 2000
  - Hurwitz Interconnect Delay Evaluation - HIDE: Programmer's Manual. CS2000-0662, Zhanhai Qin and Chung-Kuan Cheng; November 9, 2000
  - Using Annotations to Reduce Dynamic Optimization Time. CS2000-0663, Chandra Ckrintz and Brad Calder; November 16, 2000
- 2001
  - Learning and Making Decisions When Costs and Probabilities are Both Unknown. CS2001-0664, Bianca Zadrozny and Charles Elkan; January 2, 2001
  - Implementation Techniques for Efficient Data-Flow Analysis of Large Programs. CS2001-0665, Darren C. Atkinson and William G. Griswold; February 3, 2001
  - Scalable Measurement: Finding some Elephants in a Swarm of Ants. CS2001-0666, Cristian Estan and George Varghese; February 12, 2001
  - Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. CS2001-0667, Timothy Sherwood, Erez Perelman and Brad Calder; March 18, 2001
  - A Comparative Study of Two Whole Program Slicers for C. CS2001-0668, Leeann Bent, Darren C. Atkinson and William G. Griswold; April 12, 2001
  - Docking topical hierarchies: A comparison of two algorithms for reconciling keyword structures. CS2001-0669, Bryan Tower, Mark Chaisson and Richard Belew; April 26, 2001
  - Fast Content-Based Packet Handling for Intrusion Detection. CS2001-0670, Mike Fisk and George Varghese; May 7, 2001
  - Scalable Causal Message Logging for Wide-Area Environments. CS2001-0671, Karan Bhatia, Keith Marzullo and Lorenzo Alvisi; May 24, 2001
  - Reducing Load Delay to Improve Performance of Internet-Computing Programs. CS2001-0672, Chandra Krintz; May 25, 2001
  - Aggregated Bit Vector Search Algorithms for Packet Filter Lookups. CS2001-0673, Florin Baboescu and George Varghese; June 3, 2001
  - Proxy Caching with Hash Functions . CS2001-0674, Florin Baboescu; June 3, 2001
  - On-line Parallel Tomography. CS2001-0675, Shava Smallen; June 5, 2001
  - Hardware Optimizations Enabled by a Decoupled Fetch Architecture. CS2001-0676, Glenn Reinman; June 22, 2001
  - Predicting Region Branches Using Predicate Update Branch Prediction. CS2001-0677, Beth Simon, Brad Calder and Jeanne Ferrante; June 25, 2001
  - Patchable Instruction ROM Architecture. CS2001-0678, Timothy Sherwood and Brad Calder; June 25, 2001

- Rotational Position Optimization (RPO) Disk Scheduling. CS2001-0679, Walter Burkhard and John Palmer; July 16, 2001
- Improved Linux File System Hashing. CS2001-0680, Ying Chen, Walter Burkhard and John Palmer; July 16, 2001
- Minimum-Buffered Routing Of Non-critical Nets. CS2001-0681, Andrew Kahng Charles Alpert Bao Liu Ion Mandoiu Alexander Zelikovsky ; August 14, 2001
- Reengineering Cocoon with AspectJ. CS2001-0682, Leeann Bent; September 4, 2001
- Automatically Downloading Images to Improve Web Transfer Times. CS2001-0683, Girish Chandranmenon and George Varghese; September 11, 2001
- FTP-M: An FTP-like Multicast File Transfer Application. CS2001-0684, Manamohan Mysore and George Varghese; September 11, 2001
- Modifying Shortest Path Routing Protocols to Create Symmetrical Routes. CS2001-0685, Rajib Ghosh and George Varghese; September 11, 2001
- Bounded-Depth Frege with Counting Principles Polynomially Simulates Nullstellensatz Refutations. CS2001-0686, Russell Impagliazzo and Nathan Segerlind; November 14, 2001
- Guessing Two Secrets with Small Queries. CS2001-0687, Daniele Micciancio and Nathan Segerlind; November 14, 2001
- Efficient Design Space Exploration for Customized Processors. CS2001-0688, Timothy Sherwood, Mark Oskin and Brad Calder; November 20, 2001
- Relieving Register File and Instruction Window Pressure. CS2001-0689, Glenn Reinman, Brad Calder and Todd Austin; November 20, 2001
- Proof of Correctness for Sparse Tiling of Gauss-Seidel. CS2001-0690, Michelle Mills Strout, Larry Carter and Jeanne Ferrante; December 4, 2001
- Dynamic Web Stream Customizers. CS2001-0691, Jesse Steinberg and Joseph Pasquale; December 14, 2001
- A Web Middleware Architecture for Dynamic Customization of Web Content for Non-Traditional Clients. CS2001-0692, Jesse Steinberg and Joseph Pasquale; December 14, 2001
- Agent Behavior Patterns in a Wireless Internet Environment. CS2001-0693, Eugene Hung and Joseph Pasquale; December 17, 2001
- A Decoupled Predictor-Directed Stream Prefetching Architecture. CS2001-0694, Suleyman Sair, Timothy Sherwood and Brad Calder; December 17, 2001
- Turning Predicate Information to Advantage to Improve Compiler Scheduling and Branch Prediction. CS2001-0695, Beth Simon; December 27, 2001

• 2002
- Sources of Success for Information Extraction Methods. CS2002-0696, David Kauchak, Joseph Smarr and Charles Elkan; January 7, 2002
- Exponential Separation of Res(k) and Res(k+1). CS2002-0697, Sam Buss, Russell Impagliazzo and Nathan Segerlind; January 11, 2002
- A Modular Framework for Adaptive Scheduling in Grid Application Development Environments. CS2002-0698, Holly Dail; January 18, 2002
- New directions in traffic measurement and accounting. CS2002-0699, Cristian Estan and George Varghese; February 8, 2002
- Compiler and Hardware Predicated Dependency Analysis and Scheduling . CS2002-0700, Lorinda Carter; March 18, 2002
- Automatically Characterizing Large Scale Program Behavior. CS2002-0701, Timothy Sherwood, Erez Perelman and Brad Calder; March 18, 2002
- Alternatives to the k-means algorithm that find better clusterings. CS2002-0702, Greg Hamerly and Charles Elkan; April 3, 2002
- The ActiveClass Project: Experiments in Encouraging Classroom Participation. CS2002-0703, Tan Minh Truong, William G. Griswold, Matthew Ratto and Leigh Star; April 24, 2002
- ActiveCampus - Sustaining Educational Communities through Mobile Technology. CS2002-0704, William G. Griswold, Robert Boyer, Steven W. Brown, Tan Minh Truong, Ezekiel Bhasker, Gregory R. Jay and R. Benjamin Shapiro; April 24, 2002
- Counting the number of active flows on a high speed link. CS2002-0705, Cristian Estan, George Varghese and Mike Fisk; May 21, 2002
- Linear Network Reduction Via Generalized Y-Δ Transformation: Theory. CS2002-0706, Zhanhai Qin and Chung-Kuan Cheng; May 22, 2002
- The Virtual Instrument: Support for Grid-enabled Scientific Simulations. CS2002-0707, Henri Casanova, Thomas Bartol, Francine Berman, Adam and Dongarra Birnbaum Jack , Mark Ellisman, Marcio Faerman, Erhan Gockay, Michelle Miller, Graziano Obertelli, Stuart Pomerantz, Terry and Stiles Sejnowski Joel and Rich Wolski; May 31, 2002
- A Modular Scheduling Approach for Grid Application Development Environments. CS2002-0708, Holly Dail, Henri Casanova and Fran Berman; June 5, 2002
- JBIG Compression Algorithms for "Dummy Fill}± VLSI Layout Data. CS2002-0709, Robert Ellis, Andrew Kahng and Yuhong Zheng; June 14, 2002
- Phase Tracking and Prediction. CS2002-0710, Timothy Sherwood, Suleyman Sair and Brad Calder; June 23, 2002
- Optimized Trace Binaries for Architectural Evaluation. CS2002-0711, Suleyman Sair, Yuanfang Hu, Timothy Sherwood and Brad Calder; June 23, 2002
- Pointer Cache Assisted Speculative Precomputation. CS2002-0712, Jamison Collins, Suleyman Sair, Brad Calder and Dean Tullsen; June 23, 2002
- ActiveCampus - Sustaining Educational Communities through Mobile Technology. CS2002-0714, William G. Griswold, Robert Boyer, Steven W. Brown, Tan Minh Truong, Ezekiel Bhasker, Gregory R. Jay and R. Benjamin Shapiro; July 8, 2002
- The ActiveClass Project: Experiments in Encouraging Classroom Participation. CS2002-0715, Tan Minh Truong, William G. Griswold, Matthew Ratto and Susan Leigh Star; July 8, 2002
- Learning the k in k-means. CS2002-0716, Greg Hamerly and Charles Elkan; July 30, 2002
- The Y-architecture: Yet Another On-Chip Interconnect Solution. CS2002-0717, Hongyu Chen, Feng Zhou and Chung-Kuan Cheng; August 7, 2002
- Fast and Scalable Conflict Detection for Packet Classifiers. CS2002-0718, Florin Baboescu and George Varghese; August 7, 2002
- Packet Classification for Core Routers: Is there an alternatives to CAMs?. CS2002-0719, Florin Baboescu, Sumeet Singh and George Varghese; August 7, 2002
- Resource Allocation for Steerable Parallel Parameter Searches: an Experimental Study. CS2002-0720, Marcio Faerman, Adam Birnbaum, Henri Casanova and Fran Berman; August 18, 2002
- A Multi-Round Algorithm for Scheduling Divisible Workload Applications: Analysis and Experimental Evaluation . CS2002-0721, Yang Yang and Henri Casanova; September 26, 2002
- Synchronous Consensus for Dependent Process Failures. CS2002-0722, Flavio Junqueira and Keith Marzullo; October 3, 2002
- Coping with Dependent Process Failures. CS2002-0723, Flavio Junqueira, Keith Marzullo and M. Voelker Geoffrey; October 7, 2002
- Using Mobile Technology to Create Opportunitistic Interactions on a University Campus. CS2002-0724, William G. Griswold, Robert Boyer, Steven W. Brown, Tan Minh Truong, Ezekiel Bhasker, Gregory R. Jay and R. Benjamin Shapiro; October 16, 2002
- Group Membership and Wide-Area Master-Worker Computations. CS2002-0725, Kjetil Jacobsen, Xianan Zhang and Keith Marzullo; November 6, 2002
- Replication Strategies for Highly Available Peer-to-peer Storage Systems. CS2002-0726, Ranjita Bhagwan, Stefan Savage and Geoffrey M. Voelker; November 6, 2002
- Using SimPoints in Diverse Simulation Environments. CS2002-0727, Erez Perelman, Michael Van Biesbrouk, Timothy Sherwood and Brad Calder; November 16, 2002
- Interaction of Virtual Machine with the Operating System. CS2002-0728, Kiran Tati and Geoffrey M. Voelker; December 2, 2002
- Whole Page Performance. CS2002-0729, Leeann Bent and Geoffrey M. Voelker; December 16, 2002
- HYPERCUTS: A Decision Tree Based Algorithm for Fast Packet Classification. CS2002-0730, Sumeet Singh, George Varghese and Florin Baboescu; December 12, 2002
- Security in the Sanctuary System. CS2002-0731, Matthew Hohlfeld, Aditya Ojha and Bennet Yee; December 20, 2002

• 2003
- The Phoenix Recovery System: Rebuilding from the ashes of an Internet catastrophe. CS2003-0732, Flavio Junqueira, Ranjita Bhagwan, Keith Marzullo, Stefan Savage and Geoffrey M. Voelker; January 13, 2003
- Connectivity in the South American Internet. CS2003-0733, Flavio Junqueira and Renata Teixeira; January 13, 2003
- Lower Bound on the Number of Rounds for Consensus with Dependent Process Failures. CS2003-0734, Flavio Junqueira and Keith Marzullo; January 13, 2003
- MPI Process Swapping: Architecture and Experimental Verification. CS2003-0735, Otto Sievert and Henri Casanova; January 29, 2003
- Packet Classification Using Multidimensional Cutting. CS2003-0736, Sumeet Singh, Florin Baboescu, George Varghese and Jia Wang; February 7, 2003
- Consensus for Dependent Process Failures. CS2003-0737, Flavio Junqueira and Keith Marzullo; February 18, 2003
- Bitmap algorithms for counting active flows on high speed links. CS2003-0738, Cristian Estan, George Varghese and Mike Fisk; March 13, 2003
- Query Set Specification Language (QSSL). CS2003-0739, Michalis Petropoulos, Alin Deutsch and Yannis Papakonstantinou; March 24, 2003
- Increasing Object Visibility In Decentralized Unstructured Peer-To-Peer Networks Using Content Based Routing. CS2003-0740, Sumeet Singh and Florin Baboescu; March 28, 2003
- Proof of Correctness for Sparse Tiling of Gauss-Seidel. CS2003-0741, Michelle Mills Strout, Larry Carter and Jeanne Ferrante; April 1, 2003
- The case for ISP deployment of super-peers in P2P networks. CS2003-0742, Sumeet Singh, Sriram Ramabhadran, Florin Baboescu and Alex Snoeren; April 15,

2003

- On the Generalization of n>k*1. CS2003-0743, Flavio Junqueira and Keith Marzullo; April 21, 2003
- A Flow-based Task Scheduling Strategy for Distributed Systems. CS2003-0744, Sagnik Nandy, Jeanne Ferrante and Larry Carter; May 2, 2003
- Real-time Detection of Known and Unknown Worms. CS2003-0745, Sumeet Singh, Cristian Estan, George Varghese and Stefan Savage; May 30, 2003
- Automatically Inferring Patterns of Resource Consumption in Network Traffic. CS2003-0746, Cristian Estan, Stefan Savage and George Varghese; June 2, 2003
- The Measurement Manifesto. CS2003-0747, George Varghese and Cristian Estan; June 4, 2003
- Online Load Balancing and First-Hop Bandwidth Allocation in Public-Area Wireless Networks. CS2003-0748, Anand Balachandran, Sagnik Nandy, Venkat P. Rangan and Geoffrey M. Voelker; June 10, 2003
- The Impact of Address Allocation and Routing on the Structure and Implementation of Routing Tables. CS2003-0749, Harsha Narayan, Ramesh Govindan and George Varghese; June 19, 2003
- ActiveCampus - Experiments in Community-Oriented Ubiquitous Computing. CS2003-0750, William G. Griswold, Patricia Shanahan, Steven W. Brown, Robert Boyer, Matt Ratto, R. Benjamin Shapiro and Tan Minh Truong; June 24, 2003
- Bucking Free-Riders: Distributed Accounting and Settlement in Peer-to-Peer Networks. CS2003-0751, Abhishek Agrawal, Douglas Brown, Aditya Ojha and Stefan Savage; June 24, 2003
- Application-Tuned Processor Architectures. CS2003-0752, Timothy Sherwood; June 25, 2003
- Predictor-Directed Data Prefetching for Pointer-based Applications. CS2003-0753, Suleyman Sair; June 25, 2003
- Extensions to the Multi-Installment Algorithm: Affine Costs and Output Data Transfers. CS2003-0754, Yang Yang and Henri Casanova; July 16, 2003
- Cone:Augmenting DHTs to Support Distributed Resource Discovery. CS2003-0755, Ranjita Bhagwan, George Varghese and Geoffrey M. Voelker; July 21, 2003
- A Multiple Level Network Approach for Clock Skew Minimization with Process Variations. CS2003-0756, Makoto Mori, Hongyu Chen, Bo Yao and Chung-Kuan Cheng; July 28, 2003
- Structures and Algorithms for Phase Classification. CS2003-0757, Jeremy Lau, Stefan Schoenmackers and Brad Calder; July 29, 2003
- GRYD: Generalized Reduced-Order Wye-Delta Transformation: Programmer's Manual for Reduction Engine and Applications. CS2003-0758, Zhanhai Qin and Chung-Kuan Cheng; July 31, 2003
- GRYD: Generalized Reduced-Order Wye-Delta Transformation: User's Manual for Reduction Engine and Applications. CS2003-0759, Zhanhai Qin and Chung-Kuan Cheng; July 31, 2003
- Benchmark Probes for Grid Assessment. CS2003-0760, Greg Chun, Holly Dail, Henri Casanova and Allan Snavely; August 1, 2003
- The EarlyBird System for Real-time Detection of Unknown Worms. CS2003-0761, Sumeet Singh, Cristian Estan, George Varghese and Stefan Savage; August 4, 2003
- Segmentation by Example. CS2003-0762, Sameer Agarwal and Serge Belongie; August 14, 2003
- Three Brown Mice: See How They Run. CS2003-0763, Kristin Branson, Vincent Rabaud and Serge Belongie; August 19, 2003
- Approximation Methods for Thin Plate Spline Mappings and Principal Warps. CS2003-0764, Gianluca Donato and Serge Belongie; September 4, 2003
- Employing User Feedback for Fast, Accurate, Low-Maintenance Geolocationing. CS2003-0765, Ezekiel S. Bhasker, Steven W. Brown and William G. Griswold; September 8, 2003
- An Adaptive System for Real-time Summaries of Internet Traffic. CS2003-0766, Cristian Estan, Ken Keys and David Moore; September 24, 2003
- Structure from Periodic Motion. CS2003-0767, Serge Belongie and Josh Wills; October 10, 2003
- A Feature-based Approach for Determining Dense Long Range Correspondences. CS2003-0768, Josh Wills and Serge Belongie; October 20, 2003
- Characterizing and Evaluating Desktop Grids: An Empirical Study. CS2003-0769, Derrick Kondo, Michela Taufer, John Karanicolas, Charles L. Brooks, Henri Casanova and Andrew Chien; October 22, 2003
- DGMonitor: a Performance Monitoring Tool for Sandbox-based Desktop Grid Platforms. CS2003-0770, Pietro Cicotti, Michela Taufer and Andrew Chien; October 24, 2003
- A Co-Phase Matrix to Guide Simultaneous Multithreading Simulation. CS2003-0771, Michael Van Biesbrouck, Timothy Sherwood and Brad Calder; October 28, 2003
- Structures for Phase Classification. CS2003-0772, Jeremy Lau, Stefan Schoenmackers and Brad Calder; October 28, 2003
- The Entropia Virtual Machine for Desktop Grids. CS2003-0773, Brad Calder, Andrew Chien, Ju Wang and Don Yang; October 28, 2003
- Code Pointer Protection From Buffer Overflow Through Targeted Hardware Encryption. CS2003-0774, Nathan Tuck, Brad Calder and George Varghese; December 1, 2003

2004

- One Dimensional Knapsack. CS2004-0775, T. C. Hu, M. T. Shing and Leo Landa; January 14, 2004
- Using Network Flow Buffering to Improve Performance of Video over HTTP. CS2004-0776, Jesse Steinberg and Joseph Pasquale; January 14, 2004
- A Near-Optimal Algorithm for a Locality-Maximizing Placement Problem. CS2004-0777, Fan Chung, Ronald Graham, Ranjita Bhagwan, Stefan Savage and Geoffrey M. Voelker; January 16, 2004
- Tele-Reality for the Rest of Us. CS2004-0778, Neil McCurdy and William Griswold; January 16, 2004
- Critical Points for Interactive Schema Matching. CS2004-0779, Guilian Wang, Joseph Goguen, Young-Kwang Nam and Kai Lin; January 30, 2004
- Access and Mobility of Wireless PDA Users. CS2004-0780, Marvin McNett and Geoffrey M. Voelker; February 9, 2004
- Building a Hierarchy of Variable Length Intervals to Capture Hierarchical Phase Behavior. CS2004-0781, Jeremy Lau, Erez Perelman, Greg Hamerly, Timothy Sherwood and Brad Calder; March 13, 2004
- Cone: A Distributed Heap Approach to Resource Selection. CS2004-0782, Ranjita Bhagwan, Priya Mahadevan, George Varghese and Geoffrey M. Voelker; March 22, 2004
- Optimizing the Knapsack Problem. CS2004-0783, Leo Landa; April 2, 2004
- Comparison between multistage filters and sketches for finding heavy hitters. CS2004-0784, Cristian Estan; April 27, 2004
- APST-DV: Divisible Load Scheduling and Deployment on the Grid. CS2004-0785, Krijn van der Raadt, Yang Yang and Henri Casanova; April 28, 2004
- OptIPuter System Software Framework. CS2004-0786, Xinran (Ryan) Wu Andrew A. Chien Nut Taesombut Eric Weigle Huaxia Xia and Justin Burke; April 28, 2004
- Sync-scan: A fast hand-off procedure for 802.11 link layer roaming. CS2004-0787, ishwar ramani and stefan savage; May 3, 2004
- Understanding When Location-Hiding Using Overlay Networks is Feasible. CS2004-0788, Ju Wang and Andrew Chien; May 9, 2004
- Detecting Malicious Routers. CS2004-0789, Alper Mizrak, Keith Marzullo and Stefan Savage; May 24, 2004
- Semi-parametric exponential family PCA : Reducing dimensions via non-parametric latent distribution estimation. CS2004-0790, Sajama Sajama and Alon Orlitsky; June 2, 2004
- Fulcrum – An Open-Implementation Approach to Context-Aware Publish / Subscribe. CS2004-0791, Robert T. Boyer and William G. Griswold; June 8, 2004
- MobiNet: A Scalable Emulation Infrastructure for Ad Hoc and Wireless Networks. CS2004-0792, Priya Mahadevan, Adolfo Rodriguez, David Becker and Amin Vahdat; June 14, 2004
- Unified Summaries for Internet traffic. CS2004-0793, Cristian Estan; June 15, 2004
- Sage Algorithms for Knapsack Problem. CS2004-0794, Leo Landa; June 18, 2004
- Network Telescopes: Technical Report. CS2004-0795, David Moore, Colleen Shannon, Geoffrey M. Voelker and Stefan Savage; July 7, 2004
- Computing the Optimal Makespan for Jobs with Identical and Independent Tasks Scheduled on Volatile Hosts. CS2004-0796, Derrick Kondo and Henri Casanova; July 12, 2004
- Using Program Phases as Meta-Data for Runtime Energy Optimization. CS2004-0797, Cristiano Pereira and Rajesh Gupta; July 14, 2004
- Evaluation of a High Performance Erasure Code Implementation. CS2004-0798, Frank Uyeda, Huaxia Xia and Andrew A. Chien; September 13, 2004
- A New Direction in Tree Based Search Engine Architectures Using Balanced Single Port Memories. CS2004-0799, Florin Baboescu and Dean Tullsen; October

15, 2004
- Declarative Resource Naming for Macroprogramming Wireless Networks of Embedded Systems. CS2004-0800, Chalermek Intanagonwiwat, Rajesh Gupta and Amin Vahdat; November 2, 2004
- A Placement Methodology for Global Interconnect Reduction and Its Impact on Performance. CS2004-0801, Andrew Kahng, Igor Markov and Sherief Reda; October 31, 2004
- APST-DV: A Practical Framework for Scheduling and Deploying Divisible Loads on Grid Platforms. CS2004-0802, Krijn van der Raadt, Yang Yang and Henri Casanova; November 9, 2004
- Efficient Sampling Startup for Uniprocessor and Simultaneous Multithreading Simulation . CS2004-0803, Michael Van Biesbrouck, Lieven Eeckhout and Brad Calder; November 28, 2004
- Selecting Software Phase Markers with Code Structure Analysis . CS2004-0804, Jeremy Lau, Erez Perelman and Brad Calder; November 28, 2004
- Efficient Bounds Checking for C . CS2004-0805, Weihaw Chuang, Satish Narayanasamy and Brad Calder; November 28, 2004
- CLIDE: Interactively Formulating Feasible Queries on Query Rewriting-Based Systems. CS2004-0807, Michalis Petropoulos, Alin Deutsch and Yannis Papakonstantinou; December 12, 2004
- Critical-Path Aware Processor Architectures. CS2004-0808, Eric Tune; December 16, 2004
- Efficient Resource Description and High Quality Selection for Virtual Grids. CS2004-0809, Yang-Suk Kee, Dionysios Logothetis, Richard Huang, Henri Casanova and Andrew A. Chien; December 17, 2004
- Supervised dimensionality reduction using mixture models. CS2004-0810, Sajama Sajama and Alon Orlitsky; December 27, 2004
- Limit results on pattern entropy. CS2004-0811, Alon Orlitsky, Narayana Santhanam, Krishnamurthy Viswanathan and Junan Zhang; December 27, 2004

• 2005
- Weak leader election for receive-omission process failures. CS2005-0812, Flavio Junqueira and Keith Marzullo; January 26, 2005
- A Systems Architecture for Ubiquitous Video. CS2005-0813, Neil J. McCurdy and William G. Griswold; February 4, 2005
- Harnessing Mobile Ubiquitous Video. CS2005-0814, Neil J. McCurdy and William G. Griswold; February 4, 2005
- Coping with Internet catastrophes. CS2005-0816, Flavio Junqueira, Ranjita Bhagwan, Alejandro Hevia, Keith Marzullo and Geoffrey M. Voelker; February 17, 2005
- The Virtual Grid Description Language: vgDL. CS2005-0817, Andrew Chien, Henri Casanova, Yang-suk Kee and Richard Huang; February 18, 2005
- NP-Completeness of the Divisible Load Scheduling Problem on Heterogeneous Star Platforms with Affine Costs. CS2005-0818, Arnaud Legrand, Yang Yang and Henri Casanova; March 10, 2005
- Accuracy Bounds For The Scaled Bitmap Data Structure. CS2005-0819, Sumeet Singh, Cristian Estan, George Varghese and Stefan Savage; March 22, 2005
- Place-Its: Location-Based Reminders on Mobile Phones. CS2005-0820, Timothy Sohn, Kevin Li, Gunny Lee, Ian Smith, James Scott and William Griswold; March 23, 2005
- Automatic Color Calibration for Large Camera Arrays. CS2005-0821, Neel Joshi, Bennett Wilburn, Vaibhav Vaish, Marc Levoy Levoy and Mark Horowitz; May 11, 2005
- The Power of Slicing in Internet Flow Measurement. CS2005-0822, Ramana Rao Estan Kompella Cristian ; May 13, 2005
- Enhanced Design Flow and Optimizations for Multi-Project Wafers. CS2005-0823, Andrew Kahng, Ion Mandoiu, Xu Xu and Alex Zelikovsky; May 14, 2005
- The Overlay Network Content Distribution Problem. CS2005-0824, Chip Killian, Michael Vrable, Alex C. Snoeren, Amin Vahdat and Joseph Pasquale; May 18, 2005
- Combined Selection and Binding for Competitive Resource Environments. CS2005-0825, Yang-Suk Kee, Henri Casanova and Andrew A. Chien; May 18, 2005
- Evaluating Location Based Reminders. CS2005-0826, Kevin A. Li, Timothy Sohn and William G. Griswold; May 18, 2005
- Declarative Resource Naming for Macroprogramming Wireless Networks of Embedded Systems. CS2005-0827, Chalermek Intanagonwiwat, Rajesh Gupta and Amin Vahdat; May 30, 2005
- Weak Leader Election in the receive-omission failure model. CS2005-0829, Flavio Junqueira and Keith Marzullo; June 1, 2005
- Efficient Cooperative Scheduling in 802.11 Wireless Networks. CS2005-0830, Ishwar Ranani, Ramana Rao Kompella, Sriam Ramabhadran and Alex Snoren; July 7, 2005
- Coterie availability in sites (extended version). CS2005-0831, Flavio Junqueira and Keith Marzullo; July 27, 2005
- A Scalable Capstone Course for Academic Preparation. CS2005-0832, William G. Griswold; August 28, 2005
- Recognizing Cars. CS2005-0833, Louka Dlagnekov and Serge Belongie; September 28, 2005
- Maximum Instantaneous Power Estimation by Subgraph Coloring. CS2005-0834, Bao Liu; October 12, 2005
- Feedthrough Channel Effect on Wirelength Distribution in the Presence of Obstacles. CS2005-0835, Andrew Cheng Kahng Chung-Kuan Liu Bao Stroobandt Dirk ; October 12, 2005
- Charge-Matching Tail Approximation in a Piece-wise Linear-and-Exponential Function. CS2005-0836, Bao Liu; October 12, 2005
- NP-Completeness and Approximation Scheme of Zero-Skew Clock Tree Problem. CS2005-0837, Bao Liu; October 13, 2005
- Software Profiling for Deterministic Replay Debugging of User Code . CS2005-0839, Satish Narayanasamy and Brad Calder; October 18, 2005
- Automatic Logging of Operating System Effects to Simplify Application-Level Architecture Simulation. CS2005-0840, Satish Narayanasamy, Cristiano Pereira, Harish Patil, Robert Cohn and Brad Calder; October 18, 2005
- Comparing Multinomial and K-Means Clustering for SimPoint. CS2005-0841, Greg Hamerly, Erez Perelman and Brad Calder; October 20, 2005
- Peer-to-Peer Error Recovery for Hybrid Satellite-Terrestrial Networks. CS2005-0842, Eric Weigle, Matti Hiltunen, Rick Schlichting, Vinay Vaishampayan and Andrew A. Chien; October 31, 2005
- Efficient Hardware Support for Deterministic Replay Debugging of Memory Races, Interrupts and Self Modifying Code. CS2005-0843, Satish Narayanasamy, Cristiano Pereira and Brad Calder; November 14, 2005
- Detecting Phases in Parallel Applications on Shared Memory Architectures. CS2005-0844, Erez Perelman, Marzia Polito, Jean-Yves Bouguet, John Sampson, Brad Calder and Carole Dulong; November 20, 2005
- Flexible and Efficient XML Search with Complex Full-Text Predicates. CS2005-0845, Sihem Amer-Yahia, Emiran Curtmola and Alin Deutsch; December 12, 2005
- Rewriting Nested XML Queries Using Nested Views. CS2005-0846, Emiran Curtmola, Alin Deutsch, Nicola Onose and Yannis Papakonstantinou; December 12, 2005
- Partial Fault Detection Using Speculative Architecture Structures. CS2005-0847, Satish Narayanasamy, Ayse Coskun and Brad Calder; December 18, 2005
- Page-Based Transactional Memory to Provide Fast Virtual Transactions. CS2005-0848, Weihaw Chuang, Satish Narayanasamy, Gilles Pokam, Jack Sampson, Michael Van Biesbrouck, Ganesh Venkatesh, Osvaldo Colavin and Brad Calder; December 18, 2005

[Search]

NCSTRL
This server operates at UCSD Computer Science and Engineering.
Send email to webmaster@cs.ucsd.edu

EXHIBIT 10
TO DECLARATION OF
JAMES A FLIGHT

[Search]

Reports Listed by Author

- Abramson, Y.
  - Active learning for visual object detection. CS2006-0871, November 19, 2006
- Agarwal, S.
  - Segmentation by Example. CS2003-0762, August 14, 2003
- Agrawal, A.
  - Bucking Free-Riders: Distributed Accounting and Settlement in Peer-to-Peer Networks. CS2003-0751, June 24, 2003
- Albrecht, J.
  - Distributed Application Management Using Plush. CS2006-0864, July 31, 2006
- Alvisi, L.
  - Scalable Causal Message Logging for Wide-Area Networks. CS2000-0651, April 21, 2000
  - Scalable Causal Message Logging for Wide-Area Environments. CS2001-0671, May 24, 2001
- Amer-Yahia, S.
  - Flexible and Efficient XML Search with Complex Full-Text Predicates. CS2005-0845, December 12, 2005
- Andrew A.
  - OptiPuter System Software Framework. CS2004-0786, April 28, 2004
- Atkinson, D.
  - Implementation Techniques for Efficient Data-Flow Analysis of Large Programs. CS2001-0665, February 3, 2001
  - A Comparative Study of Two Whole Program Slicers for C. CS2001-0668, April 12, 2001
- Austin, T.
  - A Power Efficient Speculative Fetch Architectur. CS2000-0657, June 28, 2000
  - Relieving Register File and Instruction Window Pressure. CS2001-0689, November 20, 2001
- Baboescu, F.
  - Aggregated Bit Vector Search Algorithms for Packet Filter Lookups. CS2001-0673, June 3, 2001
  - Proxy Caching with Hash Functions . CS2001-0674, June 3, 2001
  - Fast and Scalable Conflict Detection for Packet Classifiers. CS2002-0718, August 7, 2002
  - Packet Classification for Core Routers: Is there an alternatives to CAMs?. CS2002-0719, August 7, 2002
  - HYPERCUTS: A Decision Tree Based Algorithm for Fast Packet Classification. CS2002-0730, December 12, 2002
  - Packet Classification Using Multidimensional Cutting. CS2003-0736, February 7, 2003
  - Increasing Object Visibility In Decentralized Unstructured Peer-To-Peer Networks Using Content Based Routing. CS2003-0740, March 28, 2003
  - The case for ISP deployment of super-peers in P2P networks. CS2003-0742, April 15, 2003
  - A New Direction in Tree Based Search Engine Architectures Using Balanced Single Port Memories. CS2004-0799, October 15, 2004
- Balachandran, A.
  - Online Load Balancing and First-Hop Bandwidth Allocation in Public-Area Wireless Networks. CS2003-0748, June 10, 2003
- Bartol, T.
  - The Virtual Instrument: Support for Grid-enabled Scientific Simulations. CS2002-0707, May 31, 2002
- Becker, D.
  - MobiNet: A Scalable Emulation Infrastructure for Ad Hoc and Wireless Networks. CS2004-0792, June 14, 2004
- Belew, R.
  - Docking topical hierarchies: A comparison of two algorithms for reconciling keyword structures. CS2001-0669, April 26, 2001
- Bellardo, J.
  - Jigsaw: Solving the Puzzle of Enterprise 802.11 Analysis. CS2006-0849, February 21, 2006
- Bellare, M.
  - Encode-then-encipher encryption: How to exploit nonces or redundacy in plaintexts for efficient cryptography. CS2000-0646, March 6, 2000
- Belongie, S.
  - Segmentation by Example. CS2003-0762, August 14, 2003
  - Three Brown Mice: See How They Run. CS2003-0763, August 19, 2003
  - Approximation Methods for Thin Plate Spline Mappings and Principal Warps. CS2003-0764, September 4, 2003
  - Structure from Periodic Motion. CS2003-0767, October 10, 2003
  - A Feature-based Approach for Determining Dense Long Range Correspondences. CS2003-0768, October 20, 2003
  - Recognizing Cars. CS2005-0833, September 28, 2005
- Benko, P.
  - Jigsaw: Solving the Puzzle of Enterprise 802.11 Analysis. CS2006-0849, February 21, 2006
- Bent, L.
  - A Comparative Study of Two Whole Program Slicers for C. CS2001-0668, April 12, 2001
  - Reengineering Cocoon with AspectJ. CS2001-0682, September 4, 2001
  - Whole Page Performance. CS2002-0729, December 16, 2002
- Berman, F.
  - Adaptive Performance Prediction for Distributed Data-Intensive Applications. CS1999-0619, May 18, 1999
  - Application Scheduling over Supercomputers: A Proposal. CS1999-0631, October 7, 1999
  - Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. CS1999-0632, October 14, 1999
  - Combining Workstations and Supercomputers to Support Grid Applications: The Parallel Tomography Experience. CS2000-0642, January 7, 2000
  - Application Scheduling on the Information Power Grid. CS2000-0644, January 11, 2000
  - The Virtual Instrument: Support for Grid-enabled Scientific Simulations. CS2002-0707, May 31, 2002
  - A Modular Scheduling Approach for Grid Application Development Environments. CS2002-0708, June 5, 2002
  - Resource Allocation for Steerable Parallel Parameter Searches: an Experimental Study. CS2002-0720, August 18, 2002

- Bhagwan, R.
  - Replication Strategies for Highly Available Peer-to-peer Storage Systems. CS2002-0726, November 6, 2002
  - The Phoenix Recovery System: Rebuilding from the ashes of an Internet catastrophe. CS2003-0732, January 13, 2003
  - Cone:Augmenting DHTs to Support Distributed Resource Discovery. CS2003-0755, July 21, 2003
  - A Near-Optimal Algorithm for a Locality-Maximizing Placement Problem. CS2004-0777, January 16, 2004
  - Cone: A Distributed Heap Approach to Resource Selection. CS2004-0782, March 22, 2004
  - Coping with Internet catastrophes. CS2005-0816, February 17, 2005
- Bhasker, E.
  - ActiveCampus - Sustaining Educational Communities through Mobile Technology. CS2002-0704, April 24, 2002
  - ActiveCampus - Sustaining Educational Communities through Mobile Technology. CS2002-0714, July 8, 2002
  - Using Mobile Technology to Create Opportunitistic Interactions on a University Campus. CS2002-0724, October 16, 2002
  - Employing User Feedback for Fast, Accurate, Low-Maintenance Geolocationing. CS2003-0765, September 8, 2003
- Bhatia, K.
  - Scalable Causal Message Logging for Wide-Area Networks. CS2000-0651, April 21, 2000
  - Scalable Causal Message Logging for Wide-Area Environments. CS2001-0671, May 24, 2001
- Birnbaum, A.
  - The Virtual Instrument: Support for Grid-enabled Scientific Simulations. CS2002-0707, May 31, 2002
  - Resource Allocation for Steerable Parallel Parameter Searches: an Experimental Study. CS2002-0720, August 18, 2002
- Blair, P.
  - Plane Cover Multiple Access: A New Approach to Maximizing Cellular System Capacity. CS2000-0654, May 28, 2000
- Bouguet, J.
  - Detecting Phases in Parallel Applications on Shared Memory Architectures. CS2005-0844, November 20, 2005
- Boyer, R.
  - ActiveCampus - Sustaining Educational Communities through Mobile Technology. CS2002-0704, April 24, 2002
  - ActiveCampus - Sustaining Educational Communities through Mobile Technology. CS2002-0714, July 8, 2002
  - Using Mobile Technology to Create Opportunitistic Interactions on a University Campus. CS2002-0724, October 16, 2002
  - ActiveCampus - Experiments in Community-Oriented Ubiquitous Computing. CS2003-0750, June 24, 2003
  - Fulcrum – An Open-Implementation Approach to Context-Aware Publish / Subscribe. CS2004-0791, June 8, 2004
- Branson, K.
  - Three Brown Mice: See How They Run. CS2003-0763, August 19, 2003
- Brooks, C.
  - Characterizing and Evaluating Desktop Grids: An Empirical Study. CS2003-0769, October 22, 2003
- Brown, D.
  - Bucking Free-Riders: Distributed Accounting and Settlement in Peer-to-Peer Networks. CS2003-0751, June 24, 2003
- Brown, S.
  - ActiveCampus - Sustaining Educational Communities through Mobile Technology. CS2002-0704, April 24, 2002
  - ActiveCampus - Sustaining Educational Communities through Mobile Technology. CS2002-0714, July 8, 2002
  - Using Mobile Technology to Create Opportunitistic Interactions on a University Campus. CS2002-0724, October 16, 2002
  - ActiveCampus - Experiments in Community-Oriented Ubiquitous Computing. CS2003-0750, June 24, 2003
  - Employing User Feedback for Fast, Accurate, Low-Maintenance Geolocationing. CS2003-0765, September 8, 2003
- Burke, J.
  - OptIPuter System Software Framework. CS2004-0786, April 28, 2004
- Burkhard, W.
  - Uniform Hashing with Multiple Passbits. CS2000-0658, August 18, 2000
  - Rotational Position Optimization (RPO) Disk Scheduling. CS2001-0679, July 16, 2001
  - Improved Linux File System Hashing. CS2001-0680, July 16, 2001
- Buss, S.
  - Exponential Separation of Res(k) and Res(k+1). CS2002-0697, January 11, 2002
- Calder, B.
  - Reducing the Overhead of Compilation Delay. CS2000-0648, March 27, 2000
  - Reducing DRAM Power Using Compiler Assisted Refreshing. CS2000-0649, April 21, 2000
  - Dynamic Selection of Compression Formats to Reduce Transfer Delay. CS2000-0650, April 21, 2000
  - Design Automation for Finite State Machine Predictors. CS2000-0656, June 28, 2000
  - A Power Efficient Speculative Fetch Architectur. CS2000-0657, June 28, 2000
  - Using Annotations to Reduce Dynamic Optimization Time. CS2000-0663, November 16, 2000
  - Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. CS2001-0667, March 18, 2001
  - Predicting Region Branches Using Predicate Update Branch Prediction. CS2001-0677, June 25, 2001
  - Patchable Instruction ROM Architecture. CS2001-0678, June 25, 2001
  - Efficient Design Space Exploration for Customized Processors. CS2001-0688, November 20, 2001
  - Relieving Register File and Instruction Window Pressure. CS2001-0689, November 20, 2001
  - A Decoupled Predictor-Directed Stream Prefetching Architecture. CS2001-0694, December 17, 2001
  - Automatically Characterizing Large Scale Program Behavior. CS2002-0701, March 18, 2002
  - Phase Tracking and Prediction. CS2002-0710, June 23, 2002
  - Optimized Trace Binaries for Architectural Evaluation. CS2002-0711, June 23, 2002
  - Pointer Cache Assisted Speculative Precomputation. CS2002-0712, June 23, 2002
  - Using SimPoints in Diverse Simulation Environments. CS2002-0727, November 16, 2002
  - Structures and Algorithms for Phase Classification. CS2003-0757, July 29, 2003
  - A Co-Phase Matrix to Guide Simultaneous Multithreading Simulation. CS2003-0771, October 28, 2003
  - Structures for Phase Classification. CS2003-0772, October 28, 2003
  - The Entropia Virtual Machine for Desktop Grids. CS2003-0773, October 28, 2003
  - Code Pointer Protection From Buffer Overflow Through Targeted Hardware Encryption. CS2003-0774, December 1, 2003
  - Building a Hierarchy of Variable Length Intervals to Capture Hierarchical Phase Behavior. CS2004-0781, March 13, 2004
  - Efficient Sampling Startup for Uniprocessor and Simultaneous Multithreading Simulation . CS2004-0803, November 28, 2004
  - Selecting Software Phase Markers with Code Structure Analysis . CS2004-0804, November 28, 2004
  - Efficient Bounds Checking for C . CS2004-0805, November 28, 2004
  - Software Profiling for Deterministic Replay Debugging of User Code . CS2005-0839, October 18, 2005
  - Automatic Logging of Operating System Effects to Simplify Application-Level Architecture Simulation. CS2005-0840, October 18, 2005
  - Comparing Multinomial and K-Means Clustering for SimPoint. CS2005-0841, October 20, 2005
  - Efficient Hardware Support for Deterministic Replay Debugging of Memory Races, Interrupts and Self Modifying Code. CS2005-0843, November 14, 2005
  - Detecting Phases in Parallel Applications on Shared Memory Architectures. CS2005-0844, November 20, 2005
  - Partial Fault Detection Using Speculative Architecture Structures. CS2005-0847, December 18, 2005

- Page-Based Transactional Memory to Provide Fast Virtual Transactions. CS2005-0848, December 18, 2005
- Carter, L.
  - Selecting tile shape for minimal execution time. CS1999-0616, May 20, 1999
  - Determining the idle time of a tiling. CS1999-0617, July 6, 1999
  - Proof of Correctness for Sparse Tiling of Gauss-Seidel. CS2001-0690, December 4, 2001
  - Compiler and Hardware Predicated Dependency Analysis and Scheduling . CS2002-0700, March 18, 2002
  - Proof of Correctness for Sparse Tiling of Gauss-Seidel. CS2003-0741, April 1, 2003
  - A Flow-based Task Scheduling Strategy for Distributed Systems. CS2003-0744, May 2, 2003
- Casanova, H.
  - Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. CS1999-0632, October 14, 1999
  - The Virtual Instrument: Support for Grid-enabled Scientific Simulations. CS2002-0707, May 31, 2002
  - A Modular Scheduling Approach for Grid Application Development Environments. CS2002-0708, June 5, 2002
  - Resource Allocation for Steerable Parallel Parameter Searches: an Experimental Study. CS2002-0720, August 18, 2002
  - A Multi-Round Algorithm for Scheduling Divisible Workload Applications: Analysis and Experimental Evaluation . CS2002-0721, September 26, 2002
  - MPI Process Swapping: Architecture and Experimental Verification. CS2003-0735, January 29, 2003
  - Extensions to the Multi-Installment Algorithm: Affine Costs and Output Data Transfers. CS2003-0754, July 16, 2003
  - Benchmark Probes for Grid Assessment. CS2003-0760, August 1, 2003
  - Characterizing and Evaluating Desktop Grids: An Empirical Study. CS2003-0769, October 22, 2003
  - APST-DV: Divisible Load Scheduling and Deployment on the Grid. CS2004-0785, April 28, 2004
  - Computing the Optimal Makespan for Jobs with Identical and Independent Tasks Scheduled on Volatile Hosts. CS2004-0796, July 12, 2004
  - APST-DV: A Practical Framework for Scheduling and Deploying Divisible Loads on Grid Platforms. CS2004-0802, November 9, 2004
  - Efficient Resource Description and High Quality Selection for Virtual Grids. CS2004-0809, December 17, 2004
  - The Virtual Grid Description Language: vgDL. CS2005-0817, February 18, 2005
  - NP-Completeness of the Divisible Load Scheduling Problem on Heterogeneous Star Platforms with Affine Costs. CS2005-0818, March 10, 2005
  - Combined Selection and Binding for Competitive Resource Environments. CS2005-0825, May 18, 2005
- Chaisson, M.
  - Docking topical hierarchies: A comparison of two algorithms for reconciling keyword structures. CS2001-0669, April 26, 2001
- Chandranmenon, G.
  - Automatically Downloading Images to Improve Web Transfer Times. CS2001-0683, September 11, 2001
- Charles A.
  - Minimum-Buffered Routing Of Non-critical Nets. CS2001-0681, August 14, 2001
- Chen, H.
  - The Y-architecture: Yet Another On-Chip Interconnect Solution. CS2002-0717, August 7, 2002
  - A Multiple Level Network Approach for Clock Skew Minimization with Process Variations. CS2003-0756, July 28, 2003
- Chen, Y.
  - Improved Linux File System Hashing. CS2001-0680, July 16, 2001
- Cheng, C.
  - Hurwitz Interconnect Delay Evaluation - HIDE: User's Manual. CS2000-0661, November 9, 2000
  - Hurwitz Interconnect Delay Evaluation - HIDE: Programmer's Manual. CS2000-0662, November 9, 2000
  - Linear Network Reduction Via Generalized Y-Δ Transformation: Theory. CS2002-0706, May 22, 2002
  - The Y-architecture: Yet Another On-Chip Interconnect Solution. CS2002-0717, August 7, 2002
  - A Multiple Level Network Approach for Clock Skew Minimization with Process Variations. CS2003-0756, July 28, 2003
  - GRYD: Generalized Reduced-Order Wye-Delta Transformation: Programmer's Manual for Reduction Engine and Applications. CS2003-0758, July 31, 2003
  - GRYD: Generalized Reduced-Order Wye-Delta Transformation: User's Manual for Reduction Engine and Applications. CS2003-0759, July 31, 2003
  - Fast Transient Simulation of Lossy Transmission Lines. CS2006-0874, December 13, 2006
- Cheng, Y.
  - Jigsaw: Solving the Puzzle of Enterprise 802.11 Analysis. CS2006-0849, February 21, 2006
- Chien, A.
  - Characterizing and Evaluating Desktop Grids: An Empirical Study. CS2003-0769, October 22, 2003
  - DGMonitor: a Performance Monitoring Tool for Sandbox-based Desktop Grid Platforms. CS2003-0770, October 24, 2003
  - The Entropia Virtual Machine for Desktop Grids. CS2003-0773, October 28, 2003
  - Understanding When Location-Hiding Using Overlay Networks is Feasible. CS2004-0788, May 9, 2004
  - Evaluation of a High Performance Erasure Code Implementation. CS2004-0798, September 13, 2004
  - Efficient Resource Description and High Quality Selection for Virtual Grids. CS2004-0809, December 17, 2004
  - The Virtual Grid Description Language: vgDL. CS2005-0817, February 18, 2005
  - Combined Selection and Binding for Competitive Resource Environments. CS2005-0825, May 18, 2005
  - Peer-to-Peer Error Recovery for Hybrid Satellite-Terrestrial Networks. CS2005-0842, October 31, 2005
  - RobuSTore: Robust Performance for Distributed Storage Systems. CS2006-0851, March 13, 2006
  - An Efficient General-Purpose Mechanism for Data Gathering with Accuracy Requirement in Wireless Sensor Networks. CS2006-0854, April 5, 2006
  - Failure-Resilient Expectations for Federated Systems. CS2006-0865, August 28, 2006
- Chuang, W.
  - Efficient Bounds Checking for C . CS2004-0805, November 28, 2004
  - Page-Based Transactional Memory to Provide Fast Virtual Transactions. CS2005-0848, December 18, 2005
  - Maintaining Safe Memory for Security, Debugging, and Multi-threading. CS2006-0873, December 9, 2006
- Chun, G.
  - Benchmark Probes for Grid Assessment. CS2003-0760, August 1, 2003
- Chung, F.
  - A Near-Optimal Algorithm for a Locality-Maximizing Placement Problem. CS2004-0777, January 16, 2004
- Cicotti, P.
  - DGMonitor: a Performance Monitoring Tool for Sandbox-based Desktop Grid Platforms. CS2003-0770, October 24, 2003
- Cirne, W.
  - Application Scheduling over Supercomputers: A Proposal. CS1999-0631, October 7, 1999
  - Combining Workstations and Supercomputers to Support Grid Applications: The Parallel Tomography Experience. CS2000-0642, January 7, 2000
- Ckrintz, C.
  - Using Annotations to Reduce Dynamic Optimization Time. CS2000-0663, November 16, 2000
- Cohn, R.
  - Automatic Logging of Operating System Effects to Simplify Application-Level Architecture Simulation. CS2005-0840, October 18, 2005
- Colavin, O.
  - Page-Based Transactional Memory to Provide Fast Virtual Transactions. CS2005-0848, December 18, 2005
- Collins, J.
  - Pointer Cache Assisted Speculative Precomputation. CS2002-0712, June 23, 2002

- Coskun, A.
  - Partial Fault Detection Using Speculative Architecture Structures. CS2005-0847, December 18, 2005
- Cottrell, G.
  - PCA = Gabor for Expression Recognition. CS1999-0629, October 26, 1999
- Curtmola, E.
  - Flexible and Efficient XML Search with Complex Full-Text Predicates. CS2005-0845, December 12, 2005
  - Rewriting Nested XML Queries Using Nested Views. CS2005-0846, December 12, 2005
- Dail, H.
  - A Modular Framework for Adaptive Scheduling in Grid Application Development Environments. CS2002-0698, January 18, 2002
  - A Modular Scheduling Approach for Grid Application Development Environments. CS2002-0708, June 5, 2002
  - Benchmark Probes for Grid Assessment. CS2003-0760, August 1, 2003
- Dailey, M.
  - PCA = Gabor for Expression Recognition. CS1999-0629, October 26, 1999
- Demchak, B.
  - Data Quality for Situational Awareness during Mass-Casualty Events. CS2006-0857, April 11, 2006
- Deutsch, A.
  - Query Set Specification Language (QSSL). CS2003-0739, March 24, 2003
  - CLIDE: Interactively Formulating Feasible Queries on Query Rewriting-Based Systems. CS2004-0807, December 12, 2004
  - Flexible and Efficient XML Search with Complex Full-Text Predicates. CS2005-0845, December 12, 2005
  - Rewriting Nested XML Queries Using Nested Views. CS2005-0846, December 12, 2005
  - Verification of Communicating Data-Driven Web Services. CS2006-0853, March 27, 2006
  - Data Exchange, Data Integration, and Chase. CS2006-0859, April 26, 2006
  - Privacy in GLAV Information Integration. CS2006-0869, October 16, 2006
- Diagnekov, L.
  - Recognizing Cars. CS2005-0833, September 28, 2005
- Dolev, D.
  - A Client-Server Oriented Algorithm for Virtually Synchronous Group Membership in WANs. CS1999-0623, July 7, 1999
- Donato, G.
  - Approximation Methods for Thin Plate Spline Mappings and Principal Warps. CS2003-0764, September 4, 2003
- Dulong, C.
  - Detecting Phases in Parallel Applications on Shared Memory Architectures. CS2005-0844, November 20, 2005
- Eeckhout, L.
  - Efficient Sampling Startup for Uniprocessor and Simultaneous Multithreading Simulation . CS2004-0803, November 28, 2004
- Elkan, C.
  - Learning and Making Decisions When Costs and Probabilities are Both Unknown. CS2001-0664, January 2, 2001
  - Sources of Success for Information Extraction Methods. CS2002-0696, January 7, 2002
  - Alternatives to the k-means algorithm that find better clusterings. CS2002-0702, April 3, 2002
  - Learning the k in k-means. CS2002-0716, July 30, 2002
- Ellis, R.
  - JBIG Compression Algorithms for "Dummy Fill‡ VLSI Layout Data. CS2002-0709, June 14, 2002
- Ellisman, M.
  - Combining Workstations and Supercomputers to Support Grid Applications: The Parallel Tomography Experience. CS2000-0642, January 7, 2000
  - The Virtual Instrument: Support for Grid-enabled Scientific Simulations. CS2002-0707, May 31, 2002
- Estan, C.
  - Scalable Measurement: Finding some Elephants in a Swarm of Ants. CS2001-0666, February 12, 2001
  - New directions in traffic measurement and accounting. CS2002-0699, February 8, 2002
  - Counting the number of active flows on a high speed link. CS2002-0705, May 21, 2002
  - Bitmap algorithms for counting active flows on high speed links. CS2003-0738, March 13, 2003
  - Real-time Detection of Known and Unknown Worms. CS2003-0745, May 30, 2003
  - Automatically Inferring Patterns of Resource Consumption in Network Traffic. CS2003-0746, June 2, 2003
  - The Measurement Manifesto. CS2003-0747, June 4, 2003
  - The EarlyBird System for Real-time Detection of Unknown Worms. CS2003-0761, August 4, 2003
  - An Adaptive System for Real-time Summaries of Internet Traffic. CS2003-0766, September 24, 2003
  - Comparison between multistage filters and sketches for finding heavy hitters. CS2004-0784, April 27, 2004
  - Unified Summaries for Internet traffic. CS2004-0793, June 15, 2004
  - Accuracy Bounds For The Scaled Bitmap Data Structure. CS2005-0819, March 22, 2005
- Faerman, M.
  - Adaptive Performance Prediction for Distributed Data-Intensive Applications. CS1999-0619, May 18, 1999
  - The Virtual Instrument: Support for Grid-enabled Scientific Simulations. CS2002-0707, May 31, 2002
  - Resource Allocation for Steerable Parallel Parameter Searches: an Experimental Study. CS2002-0720, August 18, 2002
- Ferrante, J.
  - Selecting tile shape for minimal execution time. CS1999-0616, May 20, 1999
  - Determining the idle time of a tiling. CS1999-0617, July 6, 1999
  - Predicting Region Branches Using Predicate Update Branch Prediction. CS2001-0677, June 25, 2001
  - Proof of Correctness for Sparse Tiling of Gauss-Seidel. CS2001-0690, December 4, 2001
  - Proof of Correctness for Sparse Tiling of Gauss-Seidel. CS2003-0741, April 1, 2003
  - A Flow-based Task Scheduling Strategy for Distributed Systems. CS2003-0744, May 2, 2003
- Fisk, M.
  - Fast Content-Based Packet Handling for Intrusion Detection. CS2001-0670, May 7, 2001
  - Counting the number of active flows on a high speed link. CS2002-0705, May 21, 2002
  - Bitmap algorithms for counting active flows on high speed links. CS2003-0738, March 13, 2003
- Freund, Y.
  - BioSpike: Efficient search for homologous proteins by indexing patterns. CS2006-0858, April 26, 2006
  - Active learning for visual object detection. CS2006-0871, November 19, 2006
- Frey, J.
  - Combining Workstations and Supercomputers to Support Grid Applications: The Parallel Tomography Experience. CS2000-0642, January 7, 2000
- Geoffrey, M.
  - Coping with Dependent Process Failures. CS2002-0723, October 7, 2002
- Ghosh, R.
  - Modifying Shortest Path Routing Protocols to Create Symmetrical Routes. CS2001-0685, September 11, 2001
- Gockay, E.

- The Virtual Instrument: Support for Grid-enabled Scientific Simulations. CS2002-0707, May 31, 2002
- Goguen, J.
  - Circular Coinduction. CS2000-0647, March 14, 2000
  - Critical Points for Interactive Schema Matching. CS2004-0779, January 30, 2004
- Govindan, R.
  - The Impact of Address Allocation and Routing on the Structure and Implementation of Routing Tables. CS2003-0749, June 19, 2003
- Graham, R.
  - A Near-Optimal Algorithm for a Locality-Maximizing Placement Problem. CS2004-0777, January 16, 2004
- Griswold, W.
  - AspectBrowser: Tool Support for Managing Dispersed Aspects. CS1999-0640, January 3, 2000
  - Teaching Software Engineering in a Compiler Project Course. CS2000-0659, September 12, 2000
  - Exploiting the Map Metaphor in a Tool for Software Evolution. CS2000-0660, September 20, 2000
  - Implementation Techniques for Efficient Data-Flow Analysis of Large Programs. CS2001-0665, February 3, 2001
  - A Comparative Study of Two Whole Program Slicers for C. CS2001-0668, April 12, 2001
  - The ActiveClass Project: Experiments in Encouraging Classroom Participation. CS2002-0703, April 24, 2002
  - ActiveCampus - Sustaining Educational Communities through Mobile Technology. CS2002-0704, April 24, 2002
  - ActiveCampus - Sustaining Educational Communities through Mobile Technology. CS2002-0714, July 8, 2002
  - The ActiveClass Project: Experiments in Encouraging Classroom Participation. CS2002-0715, July 8, 2002
  - Using Mobile Technology to Create Opportunistic Interactions on a University Campus. CS2002-0724, October 16, 2002
  - ActiveCampus - Experiments in Community-Oriented Ubiquitous Computing. CS2003-0750, June 24, 2003
  - Employing User Feedback for Fast, Accurate, Low-Maintenance Geolocationing. CS2003-0765, September 8, 2003
  - Tele-Reality for the Rest of Us. CS2004-0778, January 16, 2004
  - Fulcrum – An Open-Implementation Approach to Context-Aware Publish / Subscribe. CS2004-0791, June 8, 2004
  - A Systems Architecture for Ubiquitous Video. CS2005-0813, February 4, 2005
  - Harnessing Mobile Ubiquitous Video. CS2005-0814, February 4, 2005
  - Place-Its: Location-Based Reminders on Mobile Phones. CS2005-0820, March 23, 2005
  - Evaluating Location Based Reminders. CS2005-0826, May 18, 2005
  - A Scalable Capstone Course for Academic Preparation. CS2005-0832, August 28, 2005
  - A Robust Abstraction for First-Person Video Streaming: Techniques, Applications, and Experiments. CS2006-0855, April 7, 2006
  - Data Quality for Situational Awareness during Mass-Casualty Events. CS2006-0857, April 11, 2006
- Grove, D.
  - Reducing the Overhead of Compilation Delay. CS2000-0648, March 27, 2000
- Gupta, R.
  - Using Program Phases as Meta-Data for Runtime Energy Optimization. CS2004-0797, July 14, 2004
  - Declarative Resource Naming for Macroprogramming Wireless Networks of Embedded Systems. CS2004-0800, November 2, 2004
  - Declarative Resource Naming for Macroprogramming Wireless Networks of Embedded Systems. CS2005-0827, May 30, 2005
  - Online Learning Algorithms for Dynamic Power Management. CS2006-0856, April 8, 2006
- Hamerly, G.
  - Alternatives to the k-means algorithm that find better clusterings. CS2002-0702, April 3, 2002
  - Learning the k in k-means. CS2002-0716, July 30, 2002
  - Building a Hierarchy of Variable Length Intervals to Capture Hierarchical Phase Behavior. CS2004-0781, March 13, 2004
  - Comparing Multinomial and K-Means Clustering for SimPoint. CS2005-0841, October 20, 2005
- Hevia, A.
  - Coping with Internet catastrophes. CS2005-0816, February 17, 2005
- Hiltunen, M.
  - Peer-to-Peer Error Recovery for Hybrid Satellite-Terrestrial Networks. CS2005-0842, October 31, 2005
  - Customizable Service State Durability for Service Oriented Architectures. CS2006-0861, July 13, 2006
- Hogstedt, K.
  - Selecting tile shape for minimal execution time. CS1999-0616, May 20, 1999
  - Determining the idle time of a tiling. CS1999-0617, July 6, 1999
- Hohlfeld, M.
  - Security in the Sanctuary System. CS2002-0731, December 20, 2002
- Horowitz, M.
  - Automatic Color Calibration for Large Camera Arrays. CS2005-0821, May 11, 2005
- Hu, T.
  - Minimax Programs and Bitonic Column Matrices. CS1999-0624, June 17, 1999
  - Min Cuts Without Path Packing. CS1999-0625, June 17, 1999
  - One Dimensional Knapsack. CS2004-0775, January 14, 2004
- Hu, Y.
  - Optimized Trace Binaries for Architectural Evaluation. CS2002-0711, June 23, 2002
- Huang, R.
  - Efficient Resource Description and High Quality Selection for Virtual Grids. CS2004-0809, December 17, 2004
  - The Virtual Grid Description Language: vgDL. CS2005-0817, February 18, 2005
  - Failure-Resilient Expectations for Federated Systems. CS2006-0865, August 28, 2006
- Hung, E.
  - Agent Usage Patterns: Bridging the Gap Between Agent-Based Applications and Middleware. CS1999-0638, November 19, 1999
  - Agent Behavior Patterns in a Wireless Internet Environment. CS2001-0693, December 17, 2001
- Ie, E.
  - BioSpike: Efficient search for homologous proteins by indexing patterns. CS2006-0858, April 26, 2006
- Impagliazzo, R.
  - Bounded-Depth Frege with Counting Principles Polynomially Simulates Nullstellensatz Refutations. CS2001-0686, November 14, 2001
  - Exponential Separation of Res(k) and Res(k+1). CS2002-0697, January 11, 2002
- Intanagonwiwat, C.
  - Declarative Resource Naming for Macroprogramming Wireless Networks of Embedded Systems. CS2004-0800, November 2, 2004
  - Declarative Resource Naming for Macroprogramming Wireless Networks of Embedded Systems. CS2005-0827, May 30, 2005
- Jacobsen, K.
  - Group Membership and Wide-Area Master-Worker Computations. CS2002-0725, November 6, 2002
- Jay, G.
  - ActiveCampus - Sustaining Educational Communities through Mobile Technology. CS2002-0704, April 24, 2002
  - ActiveCampus - Sustaining Educational Communities through Mobile Technology. CS2002-0714, July 8, 2002
  - Using Mobile Technology to Create Opportunistic Interactions on a University Campus. CS2002-0724, October 16, 2002

- Jensen, H.
  - Single Image Appearance Measurement. CS2006-0868, October 13, 2006
- Joshi, N.
  - Automatic Color Calibration for Large Camera Arrays. CS2005-0821, May 11, 2005
  - Single Image Appearance Measurement. CS2006-0868, October 13, 2006
- Junqueira, F.
  - Synchronous Consensus for Dependent Process Failures. CS2002-0722, October 3, 2002
  - Coping with Dependent Process Failures. CS2002-0723, October 7, 2002
  - The Phoenix Recovery System: Rebuilding from the ashes of an Internet catastrophe. CS2003-0732, January 13, 2003
  - Connectivity in the South American Internet. CS2003-0733, January 13, 2003
  - Lower Bound on the Number of Rounds for Consensus with Dependent Process Failures. CS2003-0734, January 13, 2003
  - Consensus for Dependent Process Failures. CS2003-0737, February 18, 2003
  - On the Generalization of n>k*t. CS2003-0743, April 21, 2003
  - Weak leader election for receive-omission process failures. CS2005-0812, January 26, 2005
  - Coping with Internet catastrophes. CS2005-0816, February 17, 2005
  - Weak Leader Election in the receive-omission failure model. CS2005-0829, June 1, 2005
  - Coterie availability in sites (extended version). CS2005-0831, July 27, 2005
- Kahng, A.
  - JBIG Compression Algorithms for "Dummy Fill± VLSI Layout Data. CS2002-0709, June 14, 2002
  - A Placement Methodology for Global Interconnect Reduction and Its Impact on Performance. CS2004-0801, October 31, 2004
  - Enhanced Design Flow and Optimizations for Multi-Project Wafers. CS2005-0823, May 14, 2005
  - Feedthrough Channel Effect on Wirelength Distribution in the Presence of Obstacles. CS2005-0835, October 12, 2005
- Karanicolas, J.
  - Characterizing and Evaluating Desktop Grids: An Empirical Study. CS2003-0769, October 22, 2003
- Kato, Y.
  - AspectBrowser: Tool Support for Managing Dispersed Aspects. CS1999-0640, January 3, 2000
  - Exploiting the Map Metaphor in a Tool for Software Evolution. CS2000-0660, September 20, 2000
- Kauchak, D.
  - Sources of Success for Information Extraction Methods. CS2002-0696, January 7, 2002
- Kee, Y.
  - Efficient Resource Description and High Quality Selection for Virtual Grids. CS2004-0809, December 17, 2004
  - The Virtual Grid Description Language: vgDL. CS2005-0817, February 18, 2005
  - Combined Selection and Binding for Competitive Resource Environments. CS2005-0825, May 18, 2005
- Keidar, I.
  - A Client-Server Oriented Algorithm for Virtually Synchronous Group Membership in WANs. CS1999-0623, July 7, 1999
  - Optimistic Virtual Synchrony. CS1999-0634, November 9, 1999
- Kesselman, C.
  - Combining Workstations and Supercomputers to Support Grid Applications: The Parallel Tomography Experience. CS2000-0642, January 7, 2000
- Keys, K.
  - An Adaptive System for Real-time Summaries of Internet Traffic. CS2003-0766, September 24, 2003
- Killian, C.
  - The Overlay Network Content Distribution Problem. CS2005-0824, May 18, 2005
- Kompella, R.
  - The Power of Slicing in Internet Flow Measurement. CS2005-0822, May 13, 2005
  - Efficient Cooperative Scheduling in 802.11 Wireless Networks. CS2005-0830, July 7, 2005
- Kondo, D.
  - Characterizing and Evaluating Desktop Grids: An Empirical Study. CS2003-0769, October 22, 2003
  - Computing the Optimal Makespan for Jobs with Identical and Independent Tasks Scheduled on Volatile Hosts. CS2004-0796, July 12, 2004
- Kreibich, C.
  - Automatic Protocol Inference: Unexpected Means of Identifying Protocols. CS2006-0850, February 21, 2006
- Krintz, C.
  - Reducing the Overhead of Compilation Delay. CS2000-0648, March 27, 2000
  - Dynamic Selection of Compression Formats to Reduce Transfer Delay. CS2000-0650, April 21, 2000
  - Reducing Load Delay to Improve Performance of Internet-Computing Programs. CS2001-0672, May 25, 2001
- Landa, L.
  - One Dimensional Knapsack. CS2004-0775, January 14, 2004
  - Optimizing the Knapsack Problem. CS2004-0783, April 2, 2004
  - Sage Algorithms for Knapsack Problem. CS2004-0794, June 18, 2004
- Lau, J.
  - Structures and Algorithms for Phase Classification. CS2003-0757, July 29, 2003
  - Structures for Phase Classification. CS2003-0772, October 28, 2003
  - Building a Hierarchy of Variable Length Intervals to Capture Hierarchical Phase Behavior. CS2004-0781, March 13, 2004
  - Selecting Software Phase Markers with Code Structure Analysis . CS2004-0804, November 28, 2004
- Lee, G.
  - Place-Its: Location-Based Reminders on Mobile Phones. CS2005-0820, March 23, 2005
- Legrand, A.
  - Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. CS1999-0632, October 14, 1999
  - NP-Completeness of the Divisible Load Scheduling Problem on Heterogeneous Star Platforms with Affine Costs. CS2005-0818, March 10, 2005
- Leneri, L.
  - A Robust Abstraction for First-Person Video Streaming: Techniques, Applications, and Experiments. CS2006-0855, April 7, 2006
  - Data Quality for Situational Awareness during Mass-Casualty Events. CS2006-0857, April 11, 2006
- Levchenko, K.
  - Automatic Protocol Inference: Unexpected Means of Identifying Protocols. CS2006-0850, February 21, 2006
  - Incremental Sparse Binary Vector Similarity Search in High-Dimensional Space. CS2006-0866, September 26, 2006
- Levoy, M.
  - Automatic Color Calibration for Large Camera Arrays. CS2005-0821, May 11, 2005
- Li, K.
  - Place-Its: Location-Based Reminders on Mobile Phones. CS2005-0820, March 23, 2005
  - Evaluating Location Based Reminders. CS2005-0826, May 18, 2005
- Lieber, D.
  - Reducing the Overhead of Compilation Delay. CS2000-0648, March 27, 2000

- Lin, K.
  - Critical Points for Interactive Schema Matching. CS2004-0779, January 30, 2004
- Lin, M.
  - On the Resilience of Broadcasting Strategies in a Failure-Propogating Environment . CS1999-0610, July 6, 1999
  - Directional Gossip: Gossip in a Wide Area Network. CS1999-0622, June 16, 1999
  - Gossip versus Deterministic Flooding: Low Message Overhead and High Reliability for Broadcasting on Small Networks. CS1999-0637, November 18, 1999
- Liu, B.
  - Maximum Instantaneous Power Estimation by Subgraph Coloring. CS2005-0834, October 12, 2005
  - Charge-Matching Tail Approximation in a Piece-wise Linear-and-Exponential Function. CS2005-0836, October 12, 2005
  - NP-Completeness and Approximation Scheme of Zero-Skew Clock Tree Problem. CS2005-0837, October 13, 2005
- Logothetis, D.
  - Efficient Resource Description and High Quality Selection for Virtual Grids. CS2004-0809, December 17, 2004
  - Failure-Resilient Expectations for Federated Systems. CS2006-0865, August 28, 2006
- Ma, J.
  - Automatic Protocol Inference: Unexpected Means of Identifying Protocols. CS2006-0850, February 21, 2006
  - Incremental Sparse Binary Vector Similarity Search in High-Dimensional Space. CS2006-0866, September 26, 2006
- Ma, Z.
  - Dynamic Power Aware Packet Processing with CMP. CS2006-0852, March 21, 2006
  - Online Learning Algorithms for Dynamic Power Management. CS2006-0856, April 8, 2006
- Mahadevan, P.
  - Cone: A Distributed Heap Approach to Resource Selection. CS2004-0782, March 22, 2004
  - MobiNet: A Scalable Emulation Infrastructure for Ad Hoc and Wireless Networks. CS2004-0792, June 14, 2004
- Mandoiu, I.
  - Enhanced Design Flow and Optimizations for Multi-Project Wafers. CS2005-0823, May 14, 2005
- Markov, I.
  - A Placement Methodology for Global Interconnect Reduction and Its Impact on Performance. CS2004-0801, October 31, 2004
- Martini, P.
  - Uniform Hashing with Multiple Passbits. CS2000-0658, August 18, 2000
- Marzullo, K.
  - On the Resilience of Broadcasting Strategies in a Failure-Propogating Environment . CS1999-0610, July 6, 1999
  - Directional Gossip: Gossip in a Wide Area Network. CS1999-0622, June 16, 1999
  - A Client-Server Oriented Algorithm for Virtually Synchronous Group Membership in WANs. CS1999-0623, July 7, 1999
  - Optimistic Virtual Synchrony. CS1999-0634, November 9, 1999
  - Gossip versus Deterministic Flooding: Low Message Overhead and High Reliability for Broadcasting on Small Networks. CS1999-0637, November 18, 1999
  - Scalable Causal Message Logging for Wide-Area Networks. CS2000-0651, April 21, 2000
  - Scalable Causal Message Logging for Wide-Area Environments. CS2001-0671, May 24, 2001
  - Synchronous Consensus for Dependent Process Failures. CS2002-0722, October 3, 2002
  - Coping with Dependent Process Failures. CS2002-0723, October 7, 2002
  - Group Membership and Wide-Area Master-Worker Computations. CS2002-0725, November 6, 2002
  - The Phoenix Recovery System: Rebuilding from the ashes of an Internet catastrophe. CS2003-0732, January 13, 2003
  - Lower Bound on the Number of Rounds for Consensus with Dependent Process Failures. CS2003-0734, January 13, 2003
  - Consensus for Dependent Process Failures. CS2003-0737, February 18, 2003
  - On the Generalization of n>k*t. CS2003-0743, April 21, 2003
  - Detecting Malicious Routers. CS2004-0789, May 24, 2004
  - Weak leader election for receive-omission process failures. CS2005-0812, January 26, 2005
  - Coping with Internet catastrophes. CS2005-0816, February 17, 2005
  - Weak Leader Election in the receive-omission failure model. CS2005-0829, June 1, 2005
  - Coterie availability in sites (extended version). CS2005-0831, July 27, 2005
  - Customizable Service State Durability for Service Oriented Architectures. CS2006-0861, July 13, 2006
- Masini, S.
  - Gossip versus Deterministic Flooding: Low Message Overhead and High Reliability for Broadcasting on Small Networks. CS1999-0637, November 18, 1999
- McCurdy, N.
  - Tele-Reality for the Rest of Us. CS2004-0778, January 16, 2004
  - A Systems Architecture for Ubiquitous Video. CS2005-0813, February 4, 2005
  - Harnessing Mobile Ubiquitous Video. CS2005-0814, February 4, 2005
  - A Robust Abstraction for First-Person Video Streaming: Techniques, Applications, and Experiments. CS2006-0855, April 7, 2006
- McNett, M.
  - Access and Mobility of Wireless PDA Users. CS2004-0780, February 9, 2004
- Micciancio, D.
  - Guessing Two Secrets with Small Queries. CS2001-0687, November 14, 2001
- Miller, M.
  - The Virtual Instrument: Support for Grid-enabled Scientific Simulations. CS2002-0707, May 31, 2002
- Mizrak, A.
  - Detecting Malicious Routers. CS2004-0789, May 24, 2004
- Moore, D.
  - An Adaptive System for Real-time Summaries of Internet Traffic. CS2003-0766, September 24, 2003
  - Network Telescopes: Technical Report. CS2004-0795, July 7, 2004
- Mori, M.
  - A Multiple Level Network Approach for Clock Skew Minimization with Process Variations. CS2003-0756, July 28, 2003
- Moskovics, S.
  - Multi-Language Support in a Program Analysis and Visualization Tool. CS2000-0655, June 20, 2000
- Mysore, M.
  - FTP-M: An FTP-like Multicast File Transfer Application. CS2001-0684, September 11, 2001
- Nam, Y.
  - Critical Points for Interactive Schema Matching. CS2004-0779, January 30, 2004
- Nandy, S.
  - A Flow-based Task Scheduling Strategy for Distributed Systems. CS2003-0744, May 2, 2003
  - Online Load Balancing and First-Hop Bandwidth Allocation in Public-Area Wireless Networks. CS2003-0748, June 10, 2003
- Narayan, H.
  - The Impact of Address Allocation and Routing on the Structure and Implementation of Routing Tables. CS2003-0749, June 19, 2003
- Narayanasamy, S.

- Efficient Bounds Checking for C . CS2004-0805, November 28, 2004
- Software Profiling for Deterministic Replay Debugging of User Code . CS2005-0839, October 18, 2005
- Automatic Logging of Operating System Effects to Simplify Application-Level Architecture Simulation. CS2005-0840, October 18, 2005
- Efficient Hardware Support for Deterministic Replay Debugging of Memory Races, Interrupts and Self Modifying Code. CS2005-0843, November 14, 2005
- Partial Fault Detection Using Speculative Architecture Structures. CS2005-0847, December 18, 2005
- Page-Based Transactional Memory to Provide Fast Virtual Transactions. CS2005-0848, December 18, 2005

- Nash, A.
  - Data Exchange, Data Integration, and Chase. CS2006-0859, April 26, 2006
  - Privacy in GLAV Information Integration. CS2006-0869, October 16, 2006
- Obertelli, G.
  - The Virtual Instrument: Support for Grid-enabled Scientific Simulations. CS2002-0707, May 31, 2002
- Ojha, A.
  - Security in the Sanctuary System. CS2002-0731, December 20, 2002
  - Bucking Free-Riders: Distributed Accounting and Settlement in Peer-to-Peer Networks. CS2003-0751, June 24, 2003
- Onose, N.
  - Rewriting Nested XML Queries Using Nested Views. CS2005-0846, December 12, 2005
- Orlitsky, A.
  - Semi-parametric exponential family PCA : Reducing dimensions via non-parametric latent distribution estimation. CS2004-0790, June 2, 2004
  - Supervised dimensionality reduction using mixture models. CS2004-0810, December 27, 2004
  - Limit results on pattern entropy. CS2004-0811, December 27, 2004
- Oskin, M.
  - Efficient Design Space Exploration for Customized Processors. CS2001-0688, November 20, 2001
- Palmer, J.
  - Rotational Position Optimization (RPO) Disk Scheduling. CS2001-0679, July 16, 2001
  - Improved Linux File System Hashing. CS2001-0680, July 16, 2001
- Papakonstantinou, Y.
  - Query Set Specification Language (QSSL). CS2003-0739, March 24, 2003
  - CLIDE: Interactively Formulating Feasible Queries on Query Rewriting-Based Systems. CS2004-0807, December 12, 2004
  - Rewriting Nested XML Queries Using Nested Views. CS2005-0846, December 12, 2005
- Pasquale, J.
  - Agent Usage Patterns: Bridging the Gap Between Agent-Based Applications and Middleware. CS1999-0638, November 19, 1999
  - Limited Mobile Agents: A Practical Approach. CS2000-0641, December 29, 1999
  - Dynamic Web Stream Customizers. CS2001-0691, December 14, 2001
  - A Web Middleware Architecture for Dynamic Customization of Web Content for Non-Traditional Clients. CS2001-0692, December 14, 2001
  - Agent Behavior Patterns in a Wireless Internet Environment. CS2001-0693, December 17, 2001
  - Using Network Flow Buffering to Improve Performance of Video over HTTP. CS2004-0776, January 14, 2004
  - The Overlay Network Content Distribution Problem. CS2005-0824, May 18, 2005
- Patil, H.
  - Automatic Logging of Operating System Effects to Simplify Application-Level Architecture Simulation. CS2005-0840, October 18, 2005
- Peng, H.
  - Fast Transient Simulation of Lossy Transmission Lines. CS2006-0874, December 13, 2006
- Pereira, C.
  - Using Program Phases as Meta-Data for Runtime Energy Optimization. CS2004-0797, July 14, 2004
  - Automatic Logging of Operating System Effects to Simplify Application-Level Architecture Simulation. CS2005-0840, October 18, 2005
  - Efficient Hardware Support for Deterministic Replay Debugging of Memory Races, Interrupts and Self Modifying Code. CS2005-0843, November 14, 2005
- Perelman, E.
  - Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. CS2001-0667, March 18, 2001
  - Automatically Characterizing Large Scale Program Behavior. CS2002-0701, March 18, 2002
  - Using SimPoints in Diverse Simulation Environments. CS2002-0727, November 16, 2002
  - Building a Hierarchy of Variable Length Intervals to Capture Hierarchical Phase Behavior. CS2004-0781, March 13, 2004
  - Selecting Software Phase Markers with Code Structure Analysis . CS2004-0804, November 28, 2004
  - Comparing Multinomial and K-Means Clustering for SimPoint. CS2005-0841, October 20, 2005
  - Detecting Phases in Parallel Applications on Shared Memory Architectures. CS2005-0844, November 20, 2005
- Petropoulos, M.
  - Query Set Specification Language (QSSL). CS2003-0739, March 24, 2003
  - CLIDE: Interactively Formulating Feasible Queries on Query Rewriting-Based Systems. CS2004-0807, December 12, 2004
- Pokam, G.
  - Page-Based Transactional Memory to Provide Fast Virtual Transactions. CS2005-0848, December 18, 2005
- Polito, M.
  - Detecting Phases in Parallel Applications on Shared Memory Architectures. CS2005-0844, November 20, 2005
- Polyzos, G.
  - Plane Cover Multiple Access: A New Approach to Maximizing Cellular System Capacity. CS2000-0654, May 28, 2000
- Pomerantz, S.
  - The Virtual Instrument: Support for Grid-enabled Scientific Simulations. CS2002-0707, May 31, 2002
- Qin, Z.
  - Hurwitz Interconnect Delay Evaluation - HIDE: User's Manual. CS2000-0661, November 9, 2000
  - Hurwitz Interconnect Delay Evaluation - HIDE: Programmer's Manual. CS2000-0662, November 9, 2000
  - Linear Network Reduction Via Generalized Y-Δ Transformation: Theory. CS2002-0706, May 22, 2002
  - GRYD: Generalized Reduced-Order Wye-Delta Transformation: Programmer's Manual for Reduction Engine and Applications. CS2003-0758, July 31, 2003
  - GRYD: Generalized Reduced-Order Wye-Delta Transformation: User's Manual for Reduction Engine and Applications. CS2003-0759, July 31, 2003
- REmmel, J.
  - Data Exchange, Data Integration, and Chase. CS2006-0859, April 26, 2006
- Rabaud, V.
  - Three Brown Mice: See How They Run. CS2003-0763, August 19, 2003
- Raghavan, B.
  - Distributed Rate Limiting. CS2006-0870, October 24, 2006
- Ramabhadran, S.
  - The case for ISP deployment of super-peers in P2P networks. CS2003-0742, April 15, 2003
  - Efficient Cooperative Scheduling in 802.11 Wireless Networks. CS2005-0830, July 7, 2005
  - Distributed Rate Limiting. CS2006-0870, October 24, 2006
- Ranani, I.

- Efficient Cooperative Scheduling in 802.11 Wireless Networks, CS2005-0830, July 7, 2005
- Rangan, V.
  - Online Load Balancing and First-Hop Bandwidth Allocation in Public-Area Wireless Networks, CS2003-0748, June 10, 2003
- Rano, M.
  - The ActiveClass Project: Experiments in Encouraging Classroom Participation, CS2002-0703, April 24, 2002
  - The ActiveClass Project: Experiments in Encouraging Classroom Participation, CS2002-0715, July 8, 2002
  - ActiveCampus - Experiments in Community-Oriented Ubiquitous Computing, CS2003-0750, June 24, 2003
- Reda, S.
  - A Placement Methodology for Global Interconnect Reduction and Its Impact on Performance, CS2004-0801, October 31, 2004
- Reinman, G.
  - A Power Efficient Speculative Fetch Architecture, CS2000-0657, June 28, 2000
  - Hardware Optimizations Enabled by a Decoupled Fetch Architecture, CS2001-0676, June 22, 2001
  - Relieving Register File and Instruction Window Pressure, CS2001-0689, November 20, 2001
- Ricciardi, A.
  - On the Resilience of Broadcasting Strategies in a Failure-Propogating Environment , CS1999-0610, July 6, 1999
- Rodriguez, A.
  - MobiNet: A Scalable Emulation Infrastructure for Ad Hoc and Wireless Networks, CS2004-0792, June 14, 2004
- Rogaway, P.
  - Encode-then-encipher encryption: How to exploit nonces or redundancy in plaintexts for efficient cryptography, CS2000-0646, March 6, 2000
- Rosu, G.
  - Proofs on Safety for Untrusted Code, CS1999-0633, October 27, 1999
  - Circular Coinduction, CS2000-0647, March 14, 2000
  - Abstract Semantics for Module Composition, CS2000-0653, May 8, 2000
- Sair, S.
  - A Decoupled Predictor-Directed Stream Prefetching Architecture, CS2001-0694, December 17, 2001
  - Phase Tracking and Prediction, CS2002-0710, June 23, 2002
  - Optimized Trace Binaries for Architectural Evaluation, CS2002-0711, June 23, 2002
  - Pointer Cache Assisted Speculative Precomputation, CS2002-0712, June 23, 2002
  - Predictor-Directed Data Prefetching for Pointer-based Applications, CS2003-0753, June 25, 2003
- Sajama, S.
  - Semi-parametric exponential family PCA : Reducing dimensions via non-parametric latent distribution estimation, CS2004-0790, June 2, 2004
  - Supervised dimensionality reduction using mixture models, CS2004-0810, December 27, 2004
- Sampson, J.
  - Detecting Phases in Parallel Applications on Shared Memory Architectures, CS2005-0844, November 20, 2005
  - Page-Based Transactional Memory to Provide Fast Virtual Transactions, CS2005-0848, December 18, 2005
- Santhanam, N.
  - Limit results on pattern entropy, CS2004-0811, December 27, 2004
- Sarkar, V.
  - Reducing the Overhead of Compilation Delay, CS2000-0648, March 27, 2000
- Savage, S.
  - Replication Strategies for Highly Available Peer-to-peer Storage Systems, CS2002-0726, November 6, 2002
  - The Phoenix Recovery System: Rebuilding from the ashes of an Internet catastrophe, CS2003-0732, January 13, 2003
  - Real-time Detection of Known and Unknown Worms, CS2003-0745, May 30, 2003
  - Automatically Inferring Patterns of Resource Consumption in Network Traffic, CS2003-0746, June 2, 2003
  - Bucking Free-Riders: Distributed Accounting and Settlement in Peer-to-Peer Networks, CS2003-0751, June 24, 2003
  - The EarlyBird System for Real-time Detection of Unknown Worms, CS2003-0761, August 4, 2003
  - A Near-Optimal Algorithm for a Locality-Maximizing Placement Problem, CS2004-0777, January 16, 2004
  - Detecting Malicious Routers, CS2004-0789, May 24, 2004
  - Network Telescopes: Technical Report, CS2004-0795, July 7, 2004
  - Accuracy Bounds For The Scaled Bitmap Data Structure, CS2005-0819, March 22, 2005
  - Jigsaw: Solving the Puzzle of Enterprise 802.11 Analysis, CS2006-0849, February 21, 2006
  - Automatic Protocol Inference: Unexpected Means of Identifying Protocols, CS2006-0850, February 21, 2006
- Schlichting, R.
  - Peer-to-Peer Error Recovery for Hybrid Satellite-Terrestrial Networks, CS2005-0842, October 31, 2005
  - Customizable Service State Durability for Service Oriented Architectures, CS2006-0861, July 13, 2006
- Schoenmackers, S.
  - Structures and Algorithms for Phase Classification, CS2003-0757, July 29, 2003
  - Structures for Phase Classification, CS2003-0772, October 28, 2003
- Scott, J.
  - Place-Its: Location-Based Reminders on Mobile Phones, CS2005-0820, March 23, 2005
- Segerlind, N.
  - Proofs on Safety for Untrusted Code, CS1999-0633, October 27, 1999
  - Bounded-Depth Frege with Counting Principles Polynomially Simulates Nullstellensatz Refutations, CS2001-0686, November 14, 2001
  - Guessing Two Secrets with Small Queries, CS2001-0687, November 14, 2001
  - Exponential Separation of Res(k) and Res(k+1), CS2002-0697, January 11, 2002
- Sejnowski, T.
  - The Virtual Instrument: Support for Grid-enabled Scientific Simulations, CS2002-0707, May 31, 2002
- Shanahan, P.
  - ActiveCampus - Experiments in Community-Oriented Ubiquitous Computing, CS2003-0750, June 24, 2003
- Shannon, C.
  - Network Telescopes: Technical Report, CS2004-0795, July 7, 2004
- Shapiro, R.
  - ActiveCampus - Sustaining Educational Communities through Mobile Technology, CS2002-0704, April 24, 2002
  - ActiveCampus - Sustaining Educational Communities through Mobile Technology, CS2002-0714, July 8, 2002
  - Using Mobile Technology to Create Opportunistic Interactions on a University Campus, CS2002-0724, October 16, 2002
  - ActiveCampus - Experiments in Community-Oriented Ubiquitous Computing, CS2003-0750, June 24, 2003
- Sherwood, T.
  - Reducing DRAM Power Using Compiler Assisted Refreshing, CS2000-0649, April 21, 2000
  - Design Automation for Finite State Machine Predictors, CS2000-0656, June 28, 2000
  - Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications, CS2001-0667, March 18, 2001
  - Patchable Instruction ROM Architecture, CS2001-0678, June 25, 2001

- Efficient Design Space Exploration for Customized Processors. CS2001-0688, November 20, 2001
- A Decoupled Predictor-Directed Stream Prefetching Architecture. CS2001-0694, December 17, 2001
- Automatically Characterizing Large Scale Program Behavior. CS2002-0701, March 18, 2002
- Phase Tracking and Prediction. CS2002-0710, June 23, 2002
- Optimized Trace Binaries for Architectural Evaluation. CS2002-0711, June 23, 2002
- Using SimPoints in Diverse Simulation Environments. CS2002-0727, November 16, 2002
- Application-Tuned Processor Architectures. CS2003-0752, June 25, 2003
- A Co-Phase Matrix to Guide Simultaneous Multithreading Simulation. CS2003-0771, October 28, 2003
- Building a Hierarchy of Variable Length Intervals to Capture Hierarchical Phase Behavior. CS2004-0781, March 13, 2004

- Shing, M.
  - Min Cuts Without Path Packing. CS1999-0625, June 17, 1999
  - One Dimensional Knapsack. CS2004-0775, January 14, 2004
- Sievert, O.
  - MPI Process Swapping: Architecture and Experimental Verification. CS2003-0735, January 29, 2003
- Simon, B.
  - Predicting Region Branches Using Predicate Update Branch Prediction. CS2001-0677, June 25, 2001
  - Turning Predicate Information to Advantage to Improve Compiler Scheduling and Branch Prediction. CS2001-0695, December 27, 2001
- Singh, S.
  - Packet Classification for Core Routers: Is there an alternatives to CAMs?. CS2002-0719, August 7, 2002
  - HYPERCUTS: A Decision Tree Based Algorithm for Fast Packet Classification. CS2002-0730, December 12, 2002
  - Packet Classification Using Multidimensional Cutting. CS2003-0736, February 7, 2003
  - Increasing Object Visibility In Decentralized Unstructured Peer-To-Peer Networks Using Content Based Routing. CS2003-0740, March 28, 2003
  - The case for ISP deployment of super-peers in P2P networks. CS2003-0742, April 15, 2003
  - Real-time Detection of Known and Unknown Worms. CS2003-0745, May 30, 2003
  - The EarlyBird System for Real-time Detection of Unknown Worms. CS2003-0761, August 4, 2003
  - Accuracy Bounds For The Scaled Bitmap Data Structure. CS2005-0819, March 22, 2005
- Smallen, S.
  - Combining Workstations and Supercomputers to Support Grid Applications: The Parallel Tomography Experience. CS2000-0642, January 7, 2000
  - On-line Parallel Tomography. CS2001-0675, June 5, 2001
- Smarr, J.
  - Sources of Success for Information Extraction Methods. CS2002-0696, January 7, 2002
- Smith, I.
  - Place-Its: Location-Based Reminders on Mobile Phones. CS2005-0820, March 23, 2005
- Snavely, A.
  - Benchmark Probes for Grid Assessment. CS2003-0760, August 1, 2003
- Snoeren, A.
  - The case for ISP deployment of super-peers in P2P networks. CS2003-0742, April 15, 2003
  - The Overlay Network Content Distribution Problem. CS2005-0824, May 18, 2005
  - Jigsaw: Solving the Puzzle of Enterprise 802.11 Analysis. CS2006-0849, February 21, 2006
  - Distributed Application-Management Using Plush. CS2006-0864, July 31, 2006
  - Distributed Rate Limiting. CS2006-0870, October 24, 2006
- Snoren, A.
  - Efficient Cooperative Scheduling in 802.11 Wireless Networks. CS2005-0830, July 7, 2005
- Sohn, T.
  - Place-Its: Location-Based Reminders on Mobile Phones. CS2005-0820, March 23, 2005
  - Evaluating Location Based Reminders. CS2005-0826, May 18, 2005
- Star, L.
  - The ActiveClass Project: Experiments in Encouraging Classroom Participation. CS2002-0703, April 24, 2002
- Star, S.
  - The ActiveClass Project: Experiments in Encouraging Classroom Participation. CS2002-0715, July 8, 2002
- Steinberg, J.
  - Limited Mobile Agents: A Practical Approach. CS2000-0641, December 29, 1999
  - Dynamic Web Stream Customizers. CS2001-0691, December 14, 2001
  - A Web Middleware Architecture for Dynamic Customization of Web Content for Non-Traditional Clients. CS2001-0692, December 14, 2001
  - Using Network Flow Buffering to Improve Performance of Video over HTTP. CS2004-0776, January 14, 2004
- Strout, M.
  - Proof of Correctness for Sparse Tiling of Gauss-Seidel. CS2001-0690, December 4, 2001
  - Proof of Correctness for Sparse Tiling of Gauss-Seidel. CS2003-0741, April 1, 2003
- Su, A.
  - Adaptive Performance Prediction for Distributed Data-Intensive Applications. CS1999-0619, May 18, 1999
- Su, M.
  - Combining Workstations and Supercomputers to Support Grid Applications: The Parallel Tomography Experience. CS2000-0642, January 7, 2000
- Sugihara, R.
  - An Efficient General-Purpose Mechanism for Data Gathering with Accuracy Requirement in Wireless Sensor Networks. CS2006-0854, April 5, 2006
- Sui, L.
  - Verification of Communicating Data-Driven Web Services. CS2006-0853, March 27, 2006
- Sussman, J.
  - A Client-Server Oriented Algorithm for Virtually Synchronous Group Membership in WANs. CS1999-0623, July 7, 1999
  - Optimistic Virtual Synchrony. CS1999-0634, November 9, 1999
- Tati, K.
  - Interaction of Virtual Machine with the Operating System. CS2002-0728, December 2, 2002
  - ShortCuts: Using Soft State To Improve DHT Routing. CS2006-0862, July 27, 2006
  - Resource Reclamation in Distributed Hash Tables. CS2006-0863, July 27, 2006
- Taufer, M.
  - Characterizing and Evaluating Desktop Grids: An Empirical Study. CS2003-0769, October 22, 2003
  - DGMonitor: a Performance Monitoring Tool for Sandbox-based Desktop Grid Platforms. CS2003-0770, October 24, 2003
- Teixeira, R.
  - Connectivity in the South American Internet. CS2003-0733, January 13, 2003
- Tower, B.
  - Docking topical hierarchies: A comparison of two algorithms for reconciling keyword structures. CS2001-0669, April 26, 2001
- Truong, T.

- The ActiveClass Project: Experiments in Encouraging Classroom Participation. CS2002-0703, April 24, 2002
- ActiveCampus - Sustaining Educational Communities through Mobile Technology. CS2002-0704, April 24, 2002
- ActiveCampus - Sustaining Educational Communities through Mobile Technology. CS2002-0714, July 8, 2002
- The ActiveClass Project: Experiments in Encouraging Classroom Participation. CS2002-0715, July 8, 2002
- Using Mobile Technology to Create Opportunistic Interactions on a University Campus. CS2002-0724, October 16, 2002
- ActiveCampus - Experiments in Community-Oriented Ubiquitous Computing. CS2003-0750, June 24, 2003

- Tuck, N.
  - Code Pointer Protection From Buffer Overflow Through Targeted Hardware Encryption. CS2003-0774, December 1, 2003
- Tucker, P.
  - Minimax Programs and Bitonic Column Matrices. CS1999-0624, June 17, 1999
  - Min Cuts Without Path Packing. CS1999-0625, June 17, 1999
- Tullsen, D.
  - Pointer Cache Assisted Speculative Precomputation. CS2002-0712, June 23, 2002
  - A New Direction in Tree Based Search Engine Architectures Using Balanced Single Port Memories. CS2004-0799, October 15, 2004
- Tune, E.
  - Critical-Path Aware Processor Architectures. CS2004-0808, December 16, 2004
- Tuttle, C.
  - Distributed Application Management Using Plush. CS2006-0864, July 31, 2006
- Uyeda, F.
  - Evaluation of a High Performance Erasure Code Implementation. CS2004-0798, September 13, 2004
- Vahdat, A.
  - MobiNet: A Scalable Emulation Infrastructure for Ad Hoc and Wireless Networks. CS2004-0792, June 14, 2004
  - Declarative Resource Naming for Macroprogramming Wireless Networks of Embedded Systems. CS2004-0800, November 2, 2004
  - The Overlay Network Content Distribution Problem. CS2005-0824, May 18, 2005
  - Declarative Resource Naming for Macroprogramming Wireless Networks of Embedded Systems. CS2005-0827, May 30, 2005
  - Distributed Application Management Using Plush. CS2006-0864, July 31, 2006
- Vaish, V.
  - Automatic Color Calibration for Large Camera Arrays. CS2005-0821, May 11, 2005
- Vaishampayan, V.
  - Peer-to-Peer Error Recovery for Hybrid Satellite-Terrestrial Networks. CS2005-0842, October 31, 2005
- Van B.
  - Using SimPoints in Diverse Simulation Environments. CS2002-0727, November 16, 2002
  - A Co-Phase Matrix to Guide Simultaneous Multithreading Simulation. CS2003-0771, October 28, 2003
  - Efficient Sampling Startup for Uniprocessor and Simultaneous Multithreading Simulation . CS2004-0803, November 28, 2004
  - Page-Based Transactional Memory to Provide Fast Virtual Transactions. CS2005-0848, December 18, 2005
- Varghese, G.
  - Scalable Measurement: Finding some Elephants in a Swarm of Ants. CS2001-0666, February 12, 2001
  - Fast Content-Based Packet Handling for Intrusion Detection. CS2001-0670, May 7, 2001
  - Aggregated Bit Vector Search Algorithms for Packet Filter Lookups. CS2001-0673, June 3, 2001
  - Automatically Downloading Images to Improve Web Transfer Times. CS2001-0683, September 11, 2001
  - FTP-M: An FTP-like Multicast File Transfer Application. CS2001-0684, September 11, 2001
  - Modifying Shortest Path Routing Protocols to Create Symmetrical Routes. CS2001-0685, September 11, 2001
  - New directions in traffic measurement and accounting. CS2002-0699, February 8, 2002
  - Counting the number of active flows on a high speed link. CS2002-0705, May 21, 2002
  - Fast and Scalable Conflict Detection for Packet Classifiers. CS2002-0718, August 7, 2002
  - Packet Classification for Core Routers: Is there an alternatives to CAMs?. CS2002-0719, August 7, 2002
  - HYPERCUTS: A Decision Tree Based Algorithm for Fast Packet Classification. CS2002-0730, December 12, 2002
  - Packet Classification Using Multidimensional Cutting. CS2003-0736, February 7, 2003
  - Bitmap algorithms for counting active flows on high speed links. CS2003-0738, March 13, 2003
  - Real-time Detection of Known and Unknown Worms. CS2003-0745, May 30, 2003
  - Automatically Inferring Patterns of Resource Consumption in Network Traffic. CS2003-0746, June 2, 2003
  - The Measurement Manifesto. CS2003-0747, June 4, 2003
  - The Impact of Address Allocation and Routing on the Structure and Implementation of Routing Tables. CS2003-0749, June 19, 2003
  - Cone:Augmenting DHTs to Support Distributed Resource Discovery. CS2003-0755, July 21, 2003
  - The EarlyBird System for Real-time Detection of Unknown Worms. CS2003-0761, August 4, 2003
  - Code Pointer Protection From Buffer Overflow Through Targeted Hardware Encryption. CS2003-0774, December 1, 2003
  - Cone: A Distributed Heap Approach to Resource Selection. CS2004-0782, March 22, 2004
  - Accuracy Bounds For The Scaled Bitmap Data Structure. CS2005-0819, March 22, 2005
- Venkatesh, G.
  - Page-Based Transactional Memory to Provide Fast Virtual Transactions. CS2005-0848, December 18, 2005
- Vinnu, V.
  - Verification of Communicating Data-Driven Web Services. CS2006-0853, March 27, 2006
- Vishwanath, K.
  - Distributed Rate Limiting. CS2006-0870, October 24, 2006
- Viswanathan, K.
  - Limit results on pattern entropy. CS2004-0811, December 27, 2004
- Voelker, G.
  - Replication Strategies for Highly Available Peer-to-peer Storage Systems. CS2002-0726, November 6, 2002
  - Interaction of Virtual Machine with the Operating System. CS2002-0728, December 2, 2002
  - Whole Page Performance. CS2002-0729, December 16, 2002
  - The Phoenix Recovery System: Rebuilding from the ashes of an Internet catastrophe. CS2003-0732, January 13, 2003
  - Online Load Balancing and First-Hop Bandwidth Allocation in Public-Area Wireless Networks. CS2003-0748, June 10, 2003
  - Cone:Augmenting DHTs to Support Distributed Resource Discovery. CS2003-0755, July 21, 2003
  - A Near-Optimal Algorithm for a Locality-Maximizing Placement Problem. CS2004-0777, January 16, 2004
  - Access and Mobility of Wireless PDA Users. CS2004-0780, February 9, 2004
  - Cone: A Distributed Heap Approach to Resource Selection. CS2004-0782, March 22, 2004
  - Network Telescopes: Technical Report. CS2004-0795, July 7, 2004
  - Coping with Internet catastrophes. CS2005-0816, February 17, 2005
  - Jigsaw: Solving the Puzzle of Enterprise 802.11 Analysis. CS2006-0849, February 21, 2006
  - Automatic Protocol Inference: Unexpected Means of Identifying Protocols. CS2006-0850, February 21, 2006
  - ShortCuts: Using Soft State To Improve DHT Routing. CS2006-0862, July 27, 2006

- Resource Reclamation in Distributed Hash Tables. CS2006-0863, July 27, 2006
- Vrable, M.
  - The Overlay Network Content Distribution Problem. CS2005-0824, May 18, 2005
- Wang, G.
  - Critical Points for Interactive Schema Matching. CS2004-0779, January 30, 2004
- Wang, J.
  - Packet Classification Using Multidimensional Cutting. CS2003-0736, February 7, 2003
  - The Entropia Virtual Machine for Desktop Grids. CS2003-0773, October 28, 2003
  - Understanding When Location-Hiding Using Overlay Networks is Feasible. CS2004-0788, May 9, 2004
- Weigle, E.
  - Peer-to-Peer Error Recovery for Hybrid Satellite-Terrestrial Networks. CS2005-0842, October 31, 2005
- Wilburn, B.
  - Automatic Color Calibration for Large Camera Arrays. CS2005-0821, May 11, 2005
- Wills, J.
  - Structure from Periodic Motion. CS2003-0767, October 10, 2003
  - A Feature-based Approach for Determining Dense Long Range Correspondences. CS2003-0768, October 20, 2003
- Wolski, R.
  - Adaptive Performance Prediction for Distributed Data-Intensive Applications. CS1999-0619, May 18, 1999
  - Combining Workstations and Supercomputers to Support Grid Applications: The Parallel Tomography Experience. CS2000-0642, January 7, 2000
  - Application Scheduling on the Information Power Grid. CS2000-0644, January 11, 2000
  - The Virtual Instrument: Support for Grid-enabled Scientific Simulations. CS2002-0707, May 31, 2002
- Xia, H.
  - Evaluation of a High Performance Erasure Code Implementation. CS2004-0798, September 13, 2004
  - RobuSTore: Robust Performance for Distributed Storage Systems. CS2006-0851, March 13, 2006
- Xiao, Z.
  - Incremental Sparse Binary Vector Similarity Search in High-Dimensional Space. CS2006-0866, September 26, 2006
- Xu, X.
  - Enhanced Design Flow and Optimizations for Multi-Project Wafers. CS2005-0823, May 14, 2005
- Yang, D.
  - The Entropia Virtual Machine for Desktop Grids. CS2003-0773, October 28, 2003
- Yang, X.
  - Hurwitz Interconnect Delay Evaluation - HIDE: User's Manual. CS2000-0661, November 9, 2000
- Yang, Y.
  - A Multi-Round Algorithm for Scheduling Divisible Workload Applications: Analysis and Experimental Evaluation . CS2002-0721, September 26, 2002
  - Extensions to the Multi-Installment Algorithm: Affine Costs and Output Data Transfers. CS2003-0754, July 16, 2003
  - APST-DV: Divisible Load Scheduling and Deployment on the Grid. CS2004-0785, April 28, 2004
  - APST-DV: A Practical Framework for Scheduling and Deploying Divisible Loads on Grid Platforms. CS2004-0802, November 9, 2004
  - NP-Completeness of the Divisible Load Scheduling Problem on Heterogeneous Star Platforms with Affine Costs. CS2005-0818, March 10, 2005
- Yao, B.
  - A Multiple Level Network Approach for Clock Skew Minimization with Process Variations. CS2003-0756, July 28, 2003
- Yee, B.
  - Security in the Sanctuary System. CS2002-0731, December 20, 2002
- Yocum, K.
  - Failure-Resilient Expectations for Federated Systems. CS2006-0865, August 28, 2006
  - Distributed Rate Limiting. CS2006-0870, October 24, 2006
- Young, S.
  - Combining Workstations and Supercomputers to Support Grid Applications: The Parallel Tomography Experience. CS2000-0642, January 7, 2000
- Yuan, J.
  - AspectBrowser: Tool Support for Managing Dispersed Aspects. CS1999-0640, January 3, 2000
  - Exploiting the Map Metaphor in a Tool for Software Evolution. CS2000-0660, September 20, 2000
- Zadrozny, B.
  - Learning and Making Decisions When Costs and Probabilities are Both Unknown. CS2001-0664, January 2, 2001
- Zagorodnov, D.
  - Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. CS1999-0632, October 14, 1999
  - Application Scheduling on the Information Power Grid. CS2000-0644, January 11, 2000
- Zelikovsky, A.
  - Enhanced Design Flow and Optimizations for Multi-Project Wafers. CS2005-0823, May 14, 2005
- Zhang, J.
  - Limit results on pattern entropy. CS2004-0811, December 27, 2004
- Zhang, W.
  - Dynamic Power Aware Packet Processing with CMP. CS2006-0852, March 21, 2006
  - Event-Driven Multithreaded Dynamic Optimization. CS2006-0872, December 9, 2006
- Zhang, X.
  - Group Membership and Wide-Area Master-Worker Computations. CS2002-0725, November 6, 2002
  - Customizable Service State Durability for Service Oriented Architectures. CS2006-0861, July 13, 2006
- Zhang, Y.
  - Incremental Sparse Binary Vector Similarity Search in High-Dimensional Space. CS2006-0866, September 26, 2006
- Zheng, Y.
  - JBIG Compression Algorithms for "Dummy Fill± VLSI Layout Data. CS2002-0709, June 14, 2002
- Zhou, D.
  - Verification of Communicating Data-Driven Web Services. CS2006-0853, March 27, 2006
- Zhou, F.
  - The Y-architecture: Yet Another On-Chip Interconnect Solution. CS2002-0717, August 7, 2002

[Search]

EXHIBIT 11
TO DECLARATION OF
JAMES A FLIGHT

Automatically characterizing large scale program behavior

Full text Pdf (1.54 MB)

Source **ACM SIGOPS Operating Systems Review** archive
Volume 36 , Issue 5 (December 2002) table of contents
SESSION: System performance and optimization table of contents
Pages: 45 - 57
Year of Publication: 2002
ISBN:1-58113-574-2
Also published in ...

Authors Timothy Sherwood University of California, San Diego
Erez Perelman University of California, San Diego
Greg Hamerly University of California, San Diego
Brad Calder University of California, San Diego

Publisher ACM Press New York, NY, USA

↑ ABSTRACT

Understanding program behavior is at the foundation of computer architecture and program optimization. Many programs have wildly different behavior on even the very largest of scales (over the complete execution of the program). This realization has ramifications for many architectural and compiler techniques, from thread scheduling, to feedback directed optimizations, to the way programs are simulated. However, in order to take advantage of time-varying behavior, we must first develop the analytical tools necessary to automatically and efficiently analyze program behavior over large sections of execution.Our goal is to develop automatic techniques that are capable of finding and exploiting the *Large Scale Behavior* of programs (behavior seen over billions of instructions). The first step towards this goal is the development of a hardware independent metric that can concisely summarize the behavior of an arbitrary section of execution in a program. To this end we examine the use of *Basic Block Vectors*. We quantify the effectiveness of Basic Block Vectors in capturing program behavior across several different architectural metrics, explore the large scale behavior of several programs, and develop a set of algorithms based on clustering capable of analyzing this behavior. We then demonstrate an application of this technology to automatically determine where to simulate for a program to help guide computer architecture research.

↑ REFERENCES

Note: OCR errors may be found in this Reference List extracted from the full text article. ACM has opted to expose the complete List rather than only correct and linked references.

1 A. Ben-Dor, R. Shamir, and Z. Yakhini. Clustering gene expression patterns. Journal of Computational Biology, 6:281-297, 1999.

2 Christopher M. Bishop, Neural Networks for Pattern Recognition, Oxford University Press, Inc., New York, NY, 1995

Automatically Characterizing Large Scale Program Behavior

Timothy Sherwood Erez Perelman Greg Hamerly Brad Calder

Department of Computer Science and Engineering
University of California, San Diego

{sherwood,eperelma,ghamerly,calder}@cs.ucsd.edu

Abstract

Understanding program behavior is at the foundation of computer architecture and program optimization. Many programs have wildly different behavior on even the very largest of scales (over the complete execution of the program). This realization has ramifications for many architectural and compiler techniques, from thread scheduling, to feedback directed optimizations, to the way programs are simulated. However, in order to take advantage of time-varying behavior, we must first develop the analytical tools necessary to automatically and efficiently analyze program behavior over large sections of execution.

Our goal is to develop automatic techniques that are capable of finding and exploiting the Large Scale Behavior *of programs (behavior seen over billions of instructions). The first step towards this goal is the development of a hardware independent metric that can concisely summarize the behavior of an arbitrary section of execution in a program. To this end we examine the use of* Basic Block Vectors. *We quantify the effectiveness of Basic Block Vectors in capturing program behavior across several different architectural metrics, explore the large scale behavior of several programs, and develop a set of algorithms based on clustering capable of analyzing this behavior. We then demonstrate an application of this technology to automatically determine where to simulate for a program to help guide computer architecture research.*

1. INTRODUCTION

Programs can have wildly different behavior over their run time, and these behaviors can be seen even on the largest of scales. Understanding these large scale program behaviors can unlock many new optimizations. These range from new thread scheduling algorithms that make use of information on when a thread's behavior changes, to feedback directed optimizations targeted at not only the aggregate performance of the code but individual phases of execution, to creating simulations that accurately model full program behavior. To enable these optimizations, we must first develop the analytical tools necessary to automatically and efficiently analyze program behavior over large sections of execution.

In order to perform such an analysis we need to develop a hardware independent metric that can concisely summarize the behavior of an arbitrary section of execution in a program. In [19], we presented the use of *Basic Block Vectors* (BBV), which uses the structure of the program that is exercised during execution to determine where to simulate. A BBV represents the code blocks executed during a given interval of execution. Our goal was to find a single continuous window of executed instructions that match the whole program's execution, so that this smaller window of execution can be used for simulation instead of executing the program to completion. Using the BBVs provided us with a hardware independent way of finding this small representative window.

In this paper we examine the use of BBVs for analyzing large scale program behavior. We use BBVs to explore the large scale behavior of several programs and discover the ways in which common patterns, and code, repeat themselves over the course of execution. We quantify the effectiveness of basic block vectors in capturing this program behavior across several different architectural metrics (such as IPC, branch, and cache miss rates).

In addition to this, there is a need for a way of classifying these repeating patterns so that this information can be used for optimization. We show that this problem of classifying sections of execution is related to the problem of *clustering* from machine learning, and we develop an algorithm to quickly and effectively find these sections based on clustering. Our techniques automatically break the full execution of the program up into several sets, where the elements of each set are very similar. Once this classification is completed, analysis and optimization can be performed on a per-set basis.

We demonstrate an application of this cluster-based behavior analysis to simulation methodology for computer architecture research. By making use of clustering information we are able to accurately capture the behavior of a whole program by taking simulation results from representatives of each cluster and weighing them appropriately. This results in finding a set of simulation points that when combined accurately represents the target application and input, which in turn allows the behavior of even very complicated programs such as gcc to be captured with a small amount of simulation time. We provide simulation points (points in the program to start execution at) for Alpha binaries of all of the SPEC 2000 programs. In addition, we validate these simulation points with the IPC, branch, and cache miss rates found for complete execution of the SPEC 2000 programs.

The rest of the paper is laid out as follows. First, a summary of the methodology used in this research is described

ASPLOS X, 10/02, San Jose, CA, USA.

in Section 2. Section 3 presents a brief review of basic block vectors and an in depth look into the proposed techniques and algorithms for identifying large scale program behaviors, and an analysis of their use on several programs. Section 4 describes how clustering can be used to analyze program behavior, and describes the clustering methods used in detail. Section 5 examines the use of the techniques presented in Sections 3 and 4 on an example problem: finding where to simulate in a program to achieve results representative of full program behavior. Related work is discussed in Section 6, and the techniques presented are summarized in Section 7.

2. METHODOLOGY

In this paper we used both ATOM [21] and SimpleScalar 3.0c [3] to perform our analysis and gather our results for the Alpha AXP ISA. ATOM is used to quickly gather profiling information about the code executed for a program. SimpleScalar is used to validate the phase behavior we found when clustering our basic block profiles showing that this corresponds to the phase behavior in the programs performance and architecture metrics. The baseline microarchitecture model we simulated is detailed in Table 1. We simulate an aggressive 8-way dynamically scheduled microprocessor with a two level cache design. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch mis-prediction.

We analyze and simulated all of the SPEC 2000 benchmarks compiled for the Alpha ISA. The binaries we used in this study and how they were compiled can be found at: `http://www.simplescalar.com/`.

3. USING BASIC BLOCK VECTORS

A basic block is a section of code that is executed from start to finish with one entry and one exit. We use the frequencies with which basic blocks are executed as the metric to compare different sections of the application's execution to one another. The intuition behind this is that the behavior of the program at a given time is directly related to the code it is executing during that interval, and basic block distributions provide us with this information.

A program, when run for any interval of time, will execute each basic block a certain number of times. Knowing this information provides us with a fingerprint for that interval of execution, and tells us where in the code the application is spending its time. The basic idea is that knowing the basic block distribution for two different intervals gives us two separate fingerprints which we can then compare to find out how similar the intervals are to one another. If the fingerprints are similar, then the two intervals spend about the same amount of time in the same code, and the performance of those two intervals should be similar.

3.1 Basic Block Vector

A *Basic Block Vector* (BBV) is a single dimensional array, where there is a single element in the array for each static basic block in the program. For the results in this paper, the basic block vectors are collected in intervals of 100 million instructions throughout the execution of a program. At the end of each interval, the number of times each basic block is entered during the interval is recorded and a new count for each basic block begins for the next interval of 100 million instructions. Therefore, each element in the array is the count of how many times the corresponding basic block has been entered during an interval of execution, multiplied by the number of instructions in that basic block. By multiplying in the number of instructions in each basic block we insure that we weigh instructions the same regardless of whether they reside in a large or small basic block. We say that a Basic Block Vector which was gathered by counting basic block executions over an interval of $N \times 100$ million instructions, is a Basic Block Vector of duration N.

Because we are not interested in the actual count of basic block executions for a given interval, but rather the proportions between time spent in basic blocks, a BBV is normalized by having each element divided by the sum of all the elements in the vector.

3.2 Basic Block Vector Difference

In order to find patterns in the program we must first have some way of comparing two Basic Block Vectors. The operation we desire takes as input two Basic Block Vectors, and outputs a single number which tells us how close they are to each other. There are several ways of comparing two vectors to one another, such as taking the dot product or finding the Euclidean or Manhattan distance. In this paper we use both the Euclidean and Manhattan distances for comparing vectors.

The Euclidean distance can be found by treating each vector as a single point in D-dimensional space. The distance between two points is simply the square root of the sum of squares just as in $c^2 = a^2 + b^2$. The formula for computing the Euclidean distance of two vectors a and b in D-dimensional space is given by:

$$EuclideanDist(a,b) = \sqrt{\sum_{i=1}^{D} (a_i - b_i)^2}$$

The Manhattan distance on the other hand is the distance between two points if the only paths you can take are parallel to the axes. In two dimensions this is analogous to the distance traveled if you were to go by car through city blocks. This has the advantage that it weighs more heavily differences in each dimension (being closer in the x-dimension does not get you any closer in the y-dimension). The Manhattan distance is computed by summing the absolute value of the element-wise subtraction of two vectors. For vectors a and b in D-dimensional space, the distance can be computed as:

$$ManhattanDist(a,b) = \sum_{i=1}^{D} |a_i - b_i|$$

Because we have normalized all of the vectors, the Manhattan distance will always be a single number between 0 and 2 (because we normalize each BBV to sum to 1). This number can then be used to compare how closely related two intervals of execution are to one another. For the rest of this section we will be discussing distances in terms of Manhattan distance, because we found that it more accurately represented differences in our high-dimensional data. We present the Euclidean distance as it pertains to the clustering algorithms presented in Section 4, since it provides a more accurate representation for data with lower dimensions.

3.3 Basic Block Similarity Matrix

Now that we have a method of comparing intervals of program execution to one another, we can now concentrate on finding phase-based behavior. A phase of program behavior can be defined in several ways. Past definitions are built around the idea of a phase being a contiguous interval of exe-

Instruction Cache	8k 2-way set-associative, 32 byte blocks, 1 cycle latency
Data Cache	16k 4-way set-associative, 32 byte blocks, 2 cycle latency
Unified L2 Cache	1Meg 4-way set-associative, 32 byte blocks, 20 cycle latency
Memory	150 cycle round trip access
Branch Predictor	hybrid - 8-bit gshare w/ 8k 2-bit predictors + a 8k bimodal predictor
Out-of-Order Issue	out-of-order issue of up to 8 operations per cycle, 128 entry re-order buffer
Mechanism	load/store queue, loads may execute when all prior store addresses are known
Architecture Registers	32 integer, 32 floating point
Functional Units	8-integer ALU, 4-load/store units, 2-FP adders, 2-integer MULT/DIV, 2-FP MULT/DIV
Virtual Memory	8K byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete

Table 1: Baseline Simulation Model.

cution during which a measured program metric is relatively stable. We extend this notion of a phase to include all similar sections of execution regardless of temporal adjacency.

A key observation from this paper is that the phase behavior seen in any program metric is directly a function of the code being executed. Because of this we can use the comparison between the Basic Block Vectors as an approximate bound on how closely related any other metrics will be between those two intervals.

To find how intervals of execution relate to one another we create a *Basic Block Similarity Matrix*. The similarity matrix is an upper triangular $N \times N$ matrix, where N is the number of intervals in the program's execution. An entry at (x, y) in the matrix represents the Manhattan distance between the basic block vector at interval x and the basic block vector at interval y.

Figures 1(left and right) and 4(left) shows the similarity matrices for gzip, bzip, and gcc using the Manhattan distance. The diagonal of the matrix represents the program's execution over time from start to completion. The darker the points, the more similar the intervals are (the Manhattan distance is closer to 0), and the lighter they are the more different they are (the Manhattan distance is closer to 2).

The top left corner of each graph is the start of program execution and is the origin of the graph, $(0, 0)$, and the bottom right of the graph is the point $(N-1, N-1)$ where N is the number of intervals that the full program execution was divided up into. The way to interpret the graph is to start considering points along the diagonal axis drawn. Each point is perfectly similar to itself, so the points directly on the axis all are drawn dark. Starting from a given point on the diagonal axis of the graph, you can begin to compare how that point relates to it's neighbors forward and backward in execution by tracing horizontally or vertically. If you wish to compare a given interval x with the interval at $x + n$, you simply start at the point (x, x) on the graph and trace horizontally to the right until you reach $(x, x + n)$.

To examine the phase behavior of programs, let us first examine gzip because it has behavior on such a large scale that it is easy to see. If we examine an interval taken from 70 billion instructions into execution, in Figure 1 (left), this is directly in the middle of a large phase shown by the triangle block of dark color that surrounds this point. This means that this interval is very similar to it's neighbors both forward and backward in time. We can also see that the execution at 50 billion and 90 billion instructions is also very similar to the program behavior at 70 billion. We also note, while it may be hard to see in a printed version that the phase interval at 70 billion instructions is similar to the phases at interval 10 and 30 billion, but they are not as similar as to those around 50 and 90 billion. Compare this with the IPC and data cache miss rates for gzip shown in Figure 2. Overall, Figure 1(left) shows that the phase behavior seen in the similarity matrix lines up quite closely with the behavior of the program, with 5 large phases (the first 2 being different from the last 3) each divided by a small phase, where all of the small phases are very similar to each other.

The similarity matrix for bzip (shown on the right of Figure 1) is very interesting. Bzip has complicated behavior, with two large parts to it's execution, compression and decompression. This can readily be seen in the figure as the large dark triangular and square patches. The interesting thing about bzip is that even within each of these sections of execution there is complex behavior. This, as will be shown later, makes the behavior of bzip impossible to capture using a small contiguous section of execution.

A more complex case for finding phase behavior is gcc, which is shown on the left of Figure 4. This similarity matrix shows the results for gcc using the Manhattan distance. The similarity matrix on the right will be explained in more detail in Section 4.2.1. This figure shows that gcc does have some regular behavior. It shows that, even here, there is common code shared between sections of execution, such as the intervals around 13 billion and 36 billion. In fact the strong dark diagonal line cutting through the matrix indicates that there is good amount of repetition between offset segments of execution. By analyzing the graph we can see that interval x is very similar to interval $(x + 23.6B)$ for all x.

Figures 2 and 5 show the time varying behavior of gzip and gcc. The average IPC and data cache miss rate is shown for each 100 million interval of execution over the complete execution of the program. The time varying results graphically show the same phase behavior seen by looking at only the code executed. For example, the two phases for gcc at 13 billion and 36 billion, shown to be very similar in Figure 4, are shown to have the same IPC and data cache miss rate in Figure 5.

4. CLUSTERING

The basic block vectors provide a compact and representative summary of the program's behavior for intervals of execution. By examining the similarity between them, it is clear that there exists a high level pattern to each program's execution. In order to make use of this behavior we need to start by delineating a method of finding and representing the information. Because there are so many intervals of execution that are similar to one another, one efficient representation is to group the intervals together that have similar behavior. This problem is analogous to a *clustering* problem. Later, in Section 5, we demonstrate how we use the clusters we discover to find multiple simulation points for irregular programs or inputs like gcc. By simulating only a single representative from each cluster, we can accurately represent the whole program's execution.

4.1 Clustering Overview

The goal of clustering is to divide a set of points into groups

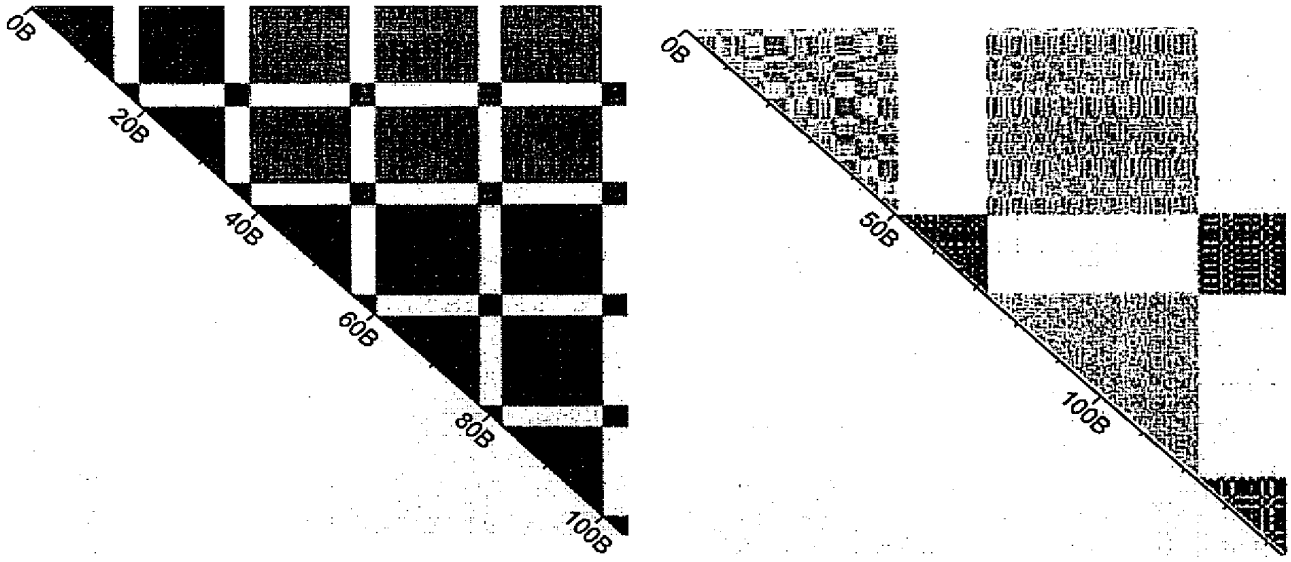

Figure 1: Basic block similarity matrix for the programs gzip-graphic (shown left) and bzip-graphic (shown right). The diagonal of the matrix represents the program's execution to completion with units in billions of instructions. The darker the points, the more similar the intervals are (the Manhattan distance is closer to 0), and the lighter the points the more different they are (the Manhattan distance is closer to 2).

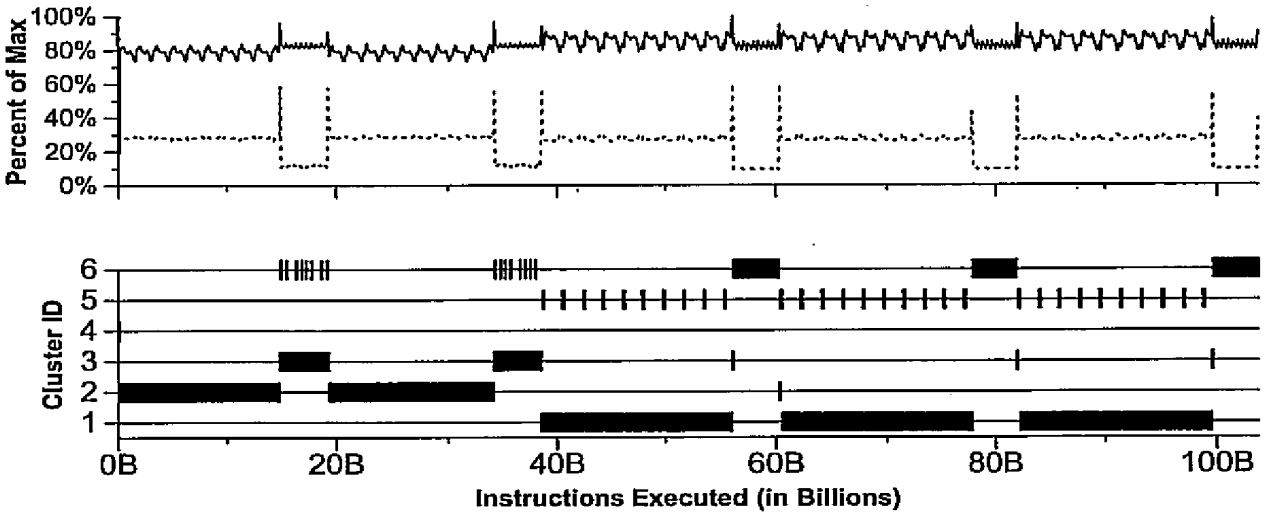

Figure 2: (top graph) Time varying graph for gzip-graphic. The average IPC (drawn with solid line) and L1 data cache miss rate (drawn with dotted line) are plotted for every interval (100 million instructions of execution) showing how these metrics vary over the program's execution. The x-axis represents the execution of the program over time, and the y-axis the percent of max value the metric had during execution. The results are non-accumulative.

Figure 3: (bottom graph) Cluster graph for gzip-graphic. The full run of the execution is partitioned into a set of 6 clusters. The x-axis is in instructions executed, and the graph shows for each interval of execution (every 100 million instructions), which cluster the interval was placed into.

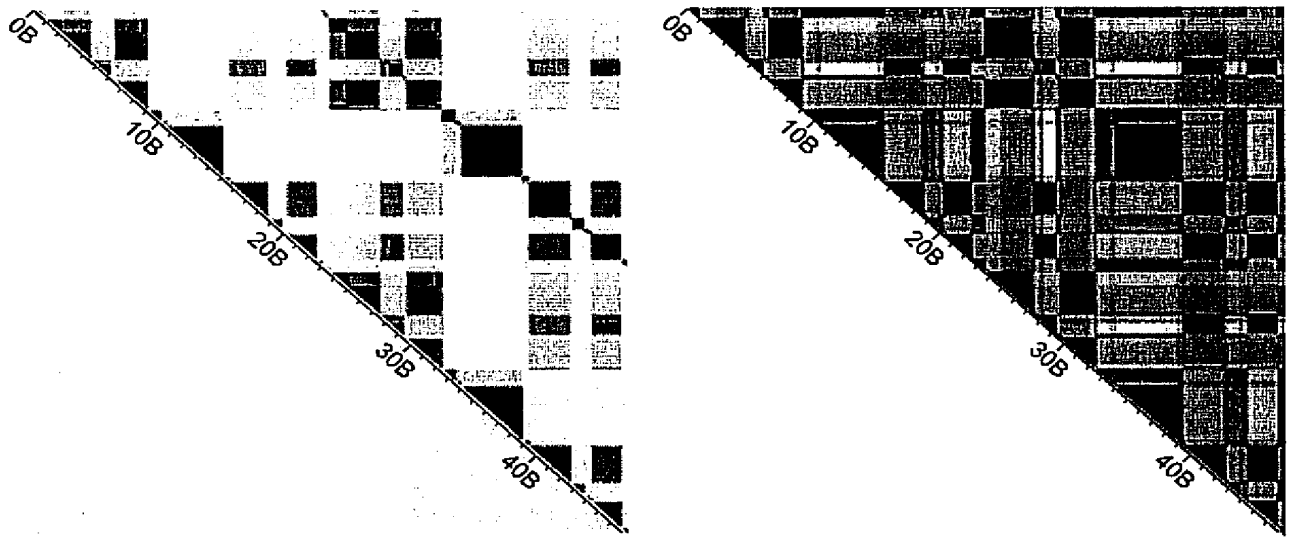

Figure 4: The original basic block similarity matrix for the program gcc (shown left), and the similarity matrix for gcc-166 drawn from projected data (on right). The figure on the left use the original basic block vectors (each of which has over 100,000 dimensions) and uses the Manhattan distance as a method of difference taking. The figure on the right uses projected data (down to 15 dimensions) and uses the Euclidean distance for difference taking.

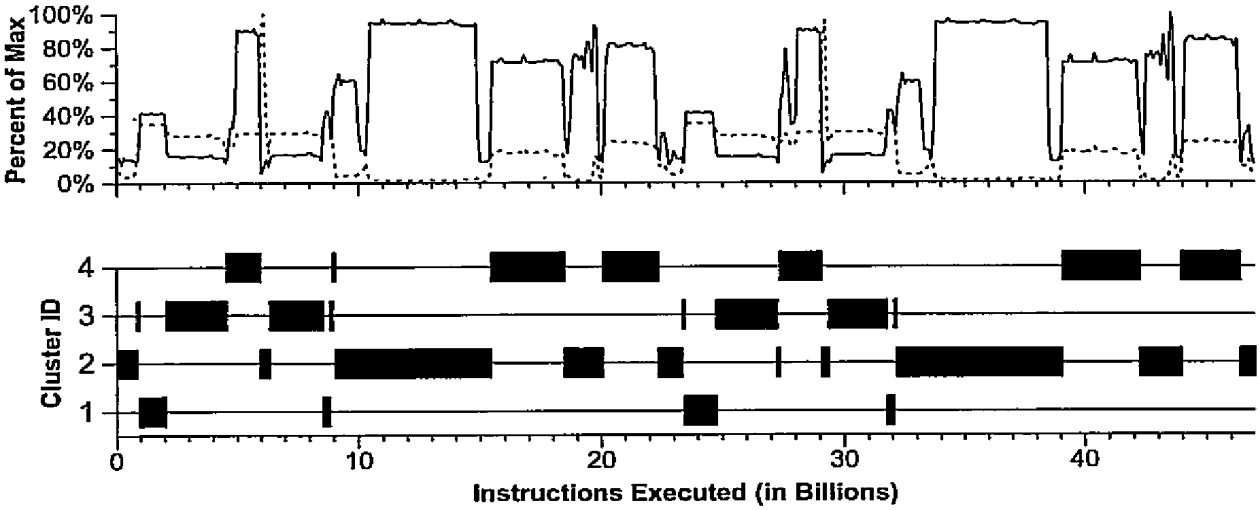

Figure 5: (top graph) Time varying graph for gcc-166. The average IPC (drawn with solid line) and L1 data cache miss rate (drawn with dotted line) are plotted for every interval (100 million instructions of execution) showing how these metrics vary over the program's execution. The x-axis represents the execution of the program over time, and the y-axis the percent of max value the metric had during execution. The results are non-accumulative.

Figure 6: (bottom graph) Cluster graph for gcc-166. The full run of the execution is partitioned into a set of 4 clusters. The x-axis is in instructions executed, and the graph shows for each interval of execution (every 100 million instructions), which cluster the interval was placed into.

such that points within each group are similar to one another (by some metric, often distance), and points in different groups are different from one another. This problem arises in other fields such as computer vision [10], document classification [22], and genomics [1], and as such it is an area of much active research. There are many clustering algorithms and many approaches to clustering. Classically, the two primary clustering approaches are Partitioning and Hierarchical:

Partitioning algorithms choose an initial solution and then use iterative updates to find a better solution. Popular algorithms such as k-means [14] and Gaussian Expectation-Maximization [2, pages 59–73] are in this family. These algorithms tend to have run time that is linear in the size of the dataset.

Hierarchical algorithms [9] either combine together similar points (called agglomerative clustering, and conceptually similar to Huffman encoding), or recursively divides the dataset into more groups (called divisive clustering). These algorithms tend to have run time that is quadratic in the size of the dataset.

4.2 Phase Finding Algorithm

For our algorithm, we use random linear projection followed by k-means. We choose to use the k-means clustering algorithm, since it is a very fast and simple algorithm that yields good results. To choose the value of k, we use the Bayesian Information Criterion (BIC) score [11, 17]. The following steps summarize our algorithm, and then several of the steps are explained in more detail:

1. Profile the basic blocks executed in each program to generate the basic block vectors for every 100 million instructions of execution.

2. Reduce the dimension of the BBV data to 15 dimensions using random linear projection.

3. Try the k-means clustering algorithm on the low-dimensional data for k values 1 to 10. Each run of k-means produces a clustering, which is a partition of the data into k different clusters.

4. For each clustering ($k = 1 \ldots 10$), score the fit of the clustering using the BIC. Choose the clustering with the smallest k, such that it's score is at least 90% as good as the best score.

4.2.1 *Random Projection*

For this clustering problem, we have to address the problem of dimensionality. All clustering algorithms suffer from the so-called "curse of dimensionality", which refers to the fact that it becomes extremely hard to cluster data as the number of dimensions increases. For the basic block vectors, the number of dimensions is the number of executed basic blocks in the program, which ranges from 2,756 to 102,038 for our experimental data, and could grow into the millions for very large programs. Another practical problem is that the running time of our clustering algorithm depends on the dimension of the data, making it slow if the dimension grows too large.

Two ways of reducing the dimension of data are dimension selection and dimension reduction. Dimension selection simply removes all but a small number of the dimensions of the data, based on a measure of goodness of each dimension for describing the data. However, this throws away a lot of data in the dimensions which are ignored. Dimension reduction reduces the number of dimensions by creating a new lower-dimensional space and then projecting each data point into the new space (where the new space's dimensions are not directly related to the old space's dimensions). This is analogous to taking a picture of 3 dimensional data at a random angle and projecting it onto a screen of 2 dimensions.

For this work we choose to use *random linear projection* [5] to create a new low-dimensional space into which we project the data. This is a simple and fast technique that is very effective at reducing the number of dimensions while retaining the properties of the data. There are two steps to reducing a dataset X (which is a matrix of basic block vectors and is of size $N_{intervals} \times D_{numbb}$, where D_{numbb} is the number of basic blocks in the program) down to D_{new} dimensions using random linear projection:

- Create a $D_{numbb} \times D_{new}$ projection matrix M by choosing a random value for each matrix entry between -1 and 1.

- Multiply X times M to obtain the new lower-dimensional dataset X' which will be of size $N_{intervals} \times D_{new}$.

For clustering programs, we found that using $D_{new} = 15$ dimensions is sufficient to still differentiate the different phases of execution. Figure 7 shows why we chose to project the data down to 15 dimensions. The graph shows the number of dimensions on the x-axis. The y-axis represents the k value found to be best on average, when the programs were projected down to the number of dimensions indicated by the x-axis. The best k is determined by the k with the highest BIC score, which is discussed in Section 4.2.3. The y-axis is shown as a percent of the maximum k seen for each program so that the curve can be examined independent of the actual number of clusters found for each program. The results show that for 15 dimensions the number of clusters found begins to stabilize and only climbs slightly. Similar results were also found using a different method of finding k in [6].

The advantages of using linear projections are twofold. First, creating new vectors with a low dimension of 15 is extremely fast and can even be done at simulation time. Secondly, using only 15 dimensions speeds up the k-means algorithm significantly, and reduces the memory requirements by several orders of magnitude over using the original basic block vectors.

Figure 4 shows the similarity matrix for gcc on the left using original BBVs, whereas the similarity matrix on the right shows the same matrix but on the data that has been projected down to 15 dimensions. For the reduced dimension data we use the Euclidean distance to measure differences, rather than the Manhattan distance used on the full data. After the projection, some information will be blurred, but overall the phases of execution that are very similar with full dimensions can still be seen to have a strong similarity with only 15 dimensions.

4.2.2 *K-means*

The k-means algorithm is an iterative optimization algorithm, which executes as two phases, which are repeated to convergence. The algorithm begins with a random assignment of k different centers, and begins its iterative process. The iterations are required because of the recursive nature of the algorithm; the cluster centers define the cluster membership for each data point, but the data point memberships define the cluster centers. Each point in the data belongs to, and can be considered a member of, a single cluster.

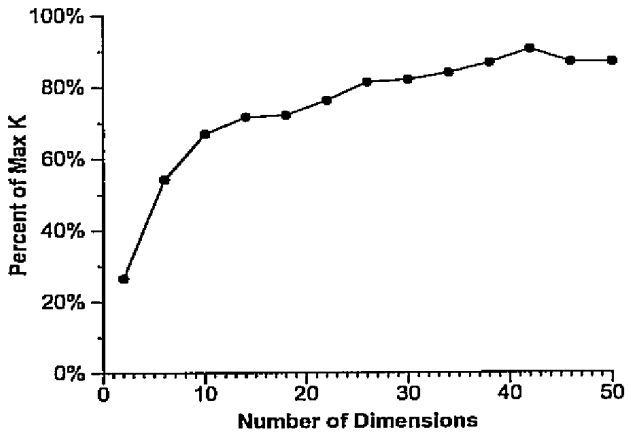

Figure 7: Motivation for random projection down to 15 dimensions (D=15). The x-axis is the number of dimensions of the projection, and the y-axis is the percent of the max number of clusters found for each program averaged over all spec programs. The results show that as you decrease the number of dimensions too far (the lowest point is two dimensions) the true clusters become collapsed on one another, and the algorithm cannot find as many clusters. By D=15 most of this effect has gone.

We initialize the k cluster centers by choosing k random points from the data to be clustered. After initialization, the k-means algorithm proceeds in two phases which are repeated until convergence:

- For each data point being clustered, compare its distance to each of the k cluster centers and assign it to (make it a member of) the cluster to which it is the closest.
- For each cluster center, change its position to the centroid of all of the points in its cluster (from the memberships just computed). The centroid is computed as the average of all the data points in the cluster.

This process is iterated until membership (and hence cluster centers) cease to change between iterations. At this point the algorithm terminates, and the output is a set of final cluster centers and a mapping of each point to the cluster that it belongs to. Since we have projected the data down to 15 dimensions, we can quickly generate the clusters for k-means with k from 1 to 10. In doing this, there are efficient algorithms for comparing the clusters that are formed for these different values of k, and choosing one that is good but still uses a small value for k is the next problem.

4.2.3 *Bayesian Information Criterion*

To compare and evaluate the different clusters formed for different k, we use the *Bayesian Information Criterion* (BIC) as a measure of the "goodness of fit" of a clustering to a dataset. More formally, the BIC is an approximation to the probability of the clustering given the data that has been clustered. Thus, the larger the BIC score, the higher the probability that the clustering being scored is a "good fit" to the data being clustered. We use the BIC formulation given in [17] for clustering with k-means, however other formulations of the BIC could also be used.

More formally, the BIC score is a penalized likelihood. There are two terms in the BIC: the likelihood and the penalty. The likelihood is a measure of how well the clustering models the data. To get the likelihood, each cluster is considered to be produced by a spherical Gaussian distribution, and the likelihood of the data in a cluster is the product of the probabilities of each point in the cluster given by the Gaussian. The likelihood for the whole dataset is just the product of the likelihoods for all clusters. However, the likelihood tends to increase without bound as more clusters are added. Therefore the second term is a penalty that offsets the likelihood growth based on the number of clusters. The BIC is formulated as

$$BIC(D,k) = l(D|k) - \frac{p_j}{2}\log(R)$$

where $l(D|k)$ is the likelihood, R is the number of points in the data, and p_j is the number of parameters to estimate, which is $(k-1)+dk+1$ for $(k-1)$ cluster probabilities, k cluster center estimates which each require d dimensions, and 1 variance estimate. To compute $l(D|k)$ we use

$$l(D|k) = \sum_{i=1}^{k} -\frac{R_i}{2}\log(2\pi) - \frac{R_i d}{2}\log(\sigma^2) - \frac{R_i - 1}{2} + R_i\log(R_i/R)$$

where R_i is the number of points in the ith cluster, and σ^2 is the average variance of the Euclidean distance from each point to its cluster center.

For a given program and inputs, the BIC score is calculated for each k-means clustering, for k from 1 to N. We then choose the clustering that achieves a BIC score that is at least 90% of the spread between the largest and smallest BIC score that the algorithm has seen. Figure 8 shows the benefit of choosing a BIC with a high value and its relationship with the variance in IPC seen for that cluster. The y-axis shows the percent of IPC variance seen for a given clustering, and the corresponding BIC score the clustering received. Each point on the graph represents the average or max IPC variance for all points in the range of ±5% of the BIC score shown. The results show that picking clusterings that represent greater than 80% of the BIC score resulted in an IPC variance of less than 20% on average. The IPC variance was computed as the weighted sum of the IPC variance for each cluster, where the weight for a cluster is the number of points in that cluster. The IPC variance for each cluster is simply the variance of the IPC for all the points in that cluster.

4.3 Clusters and Phase Behavior

Figures 3 and 6 show the 6 clusters formed for gzip and the 4 clusters formed for gcc. The X-axis corresponds to the execution of the program in billions of instructions, and each interval (each of 100 million instructions) is tagged to be in one of the N clusters (labeled on the Y-axis). These figures, just as for Figures 1 and 4, show the execution of the programs to completion.

For gzip, the full run of the execution is partitioned into a set of 6 clusters. Looking to Figure 1(left) for comparison, we see that the cluster behavior captured by our tool lines up quite closely with the behavior of the program. The majority of the points are contained by clusters 1,2,3 and 6. Clusters 1 and 2 represent the large sections of execution which are similar to one another. Clusters 3 and 6 capture the smaller phases which lie in between these large phases, while cluster 5 contains a small subset of the larger phases, and cluster 4 represents the initialization phase.

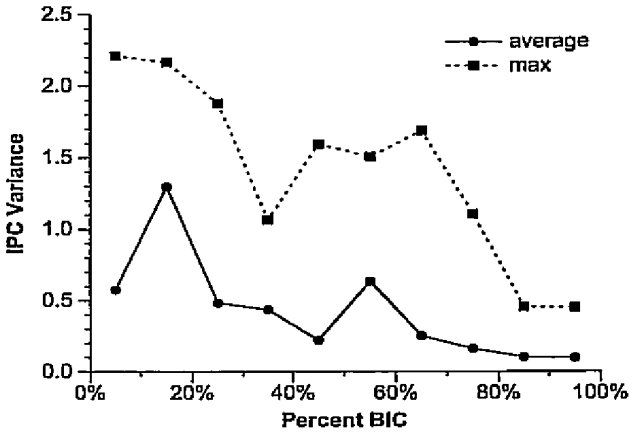

Figure 8: Plot of average IPC variance and max IPC variance versus the BIC. These results indicate that for our data, a clustering found to have a BIC score greater than 80% will have, on average, and IPC variance of less than 0.2.

In the cluster graph for gcc, shown in Figure 6, the run is now partitioned into 4 different clusters. Looking to Figure 4 for comparison, we see that even the more complicated behavior of gcc is captured correctly by our tool. Clusters 2 and 4 correspond to the dark boxes shown parallel to the diagonal axis. It should also be noted that the projection does introduce some degree of error into the clustering. For example, the first group of points in cluster 2 are not really that similar to the other points in the cluster. Comparing the two similarity matrices in Figure 4, shows the introduction of a dark band at (0,30) on the graph which was not in the original (un-projected) data. Despite these small errors, the clustering is still very good, and the impact of any such errors will be minimized in the next section.

5. FINDING SIMULATION POINTS

Modern computer architecture research relies heavily on cycle accurate simulation to help evaluate new architectural features. While the performance of processors continues to grow exponentially, the amount of complexity within a processor continues to grow at an even a faster rate. With each generation of processor more transistors are added, and more things are done in parallel on chip in a given cycle while at the same time cycle times continue to decrease. This growing gap between speed and complexity means that the time to simulate a constant amount of processor time is growing. It is already to the point that executing programs fully to completion in a detailed simulator is no longer feasible for architectural studies. Since detailed simulation takes a great deal of processing power, only a small subset of a whole program can be simulated.

SimpleScalar [3], one of the faster cycle-level simulators, can simulate around 400 million instructions per hour. Unfortunately many of the new SPEC 2000 programs execute for 300 billion instructions or more. At 400 million instructions per hour this will take approximately 1 month of CPU time.

Because it is only feasible to execute a small portion of the program, it is very important that the section simulated is an accurate representation of the program's behavior as a whole. The basic block vector and cluster analysis presented in Sections 3 and 4 will allow us to make sure that this is the case.

5.1 Single Simulation Points

In [19], we used basic block vectors to automatically find a single simulation point to potentially represent the complete execution of a program. A *Simulation Point* is a starting simulation place (in number of instructions executed from the start of execution) in a program's execution derived from our analysis. That algorithm creates a target basic block vector, which is a BBV that represents the complete execution of the program. The Manhattan distance between each interval BBV and the target BBV is computed. The BBV with the lowest Manhattan distance represents the single simulation point that executes the code closest to the complete execution of the program. This approach is used to calculate the long single simulation points (LongSP) described below.

In comparison, the single simulation point results in this paper are calculated by choosing the BBV that has the smallest Euclidean distance from the centroid of the whole dataset in the 15-dimensional space, a method which we find superior to the original method. The 15-dimensional centroid is formed by taking the average of each dimension over all intervals in the cluster.

Figure 9 shows the IPC estimated by executing only a single interval, all 100 million instructions long but chosen by different methods, for all SPEC 2000 programs. This is shown in comparison to the IPC found by executing the program to completion. The results are from SimpleScalar using the architecture model described in Section 2, and all fast forwarding is done so that all of the architecture structures are completely warmed up when starting simulation (no cold-start effect).

The first bar, labeled none, is the IPC found when executing only the first 100 million instructions from the start of execution (without any fast forwarding). The second bar, `FF-Billion` shows the results after blindly fast forwarding 1 billion instructions before starting simulation. The third bar, `SimPoint` shows the IPC using our single simulation point analysis described above, and the last bar shows the IPC of simulating the program to completion (labeled `Full`). Because these are actual IPC values, values which are closer to the `Full` bar are better.

The results in Figure 9 shows that the single simulation points are very close to the actual full execution of the program, especially when compared against the ad-hoc techniques. Starting simulation at the start of the program results in an average error of 210%, whereas blindly fast forwarding results in an average 80% IPC error. Using our single simulation point analysis we reduce the average IPC error to 18%. These results show that it is possible to reasonably capture the behavior of the most programs using a very small slice of execution.

Table 2 shows the actual simulation points chosen along with the program counter (PC) and procedure name corresponding to the start of the interval. If an input is not attached to the program name, then the default ref input was used. Columns 2 through 4 are in terms of the number of intervals (each 100 million instruction long). The first column is the number of instructions executed by the program, on the specific input, when run to completion. The second column shows the end of initialization phase calculated as described in [19]. The third column shows the single simulation point automatically chosen as described above. This simu-

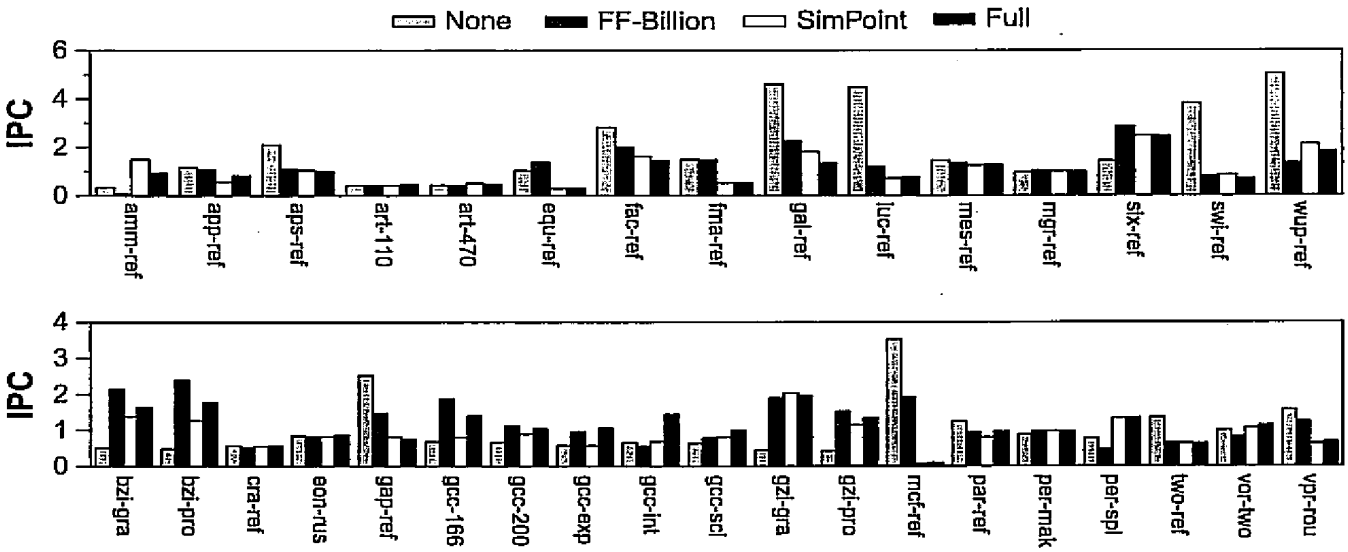

Figure 9: Simulation results starting simulation at the start of the program (none), blindly fast forwarding 1 billion instructions, using a single simulation point, and the IPC of the full execution of the program.

lation point is used to fast forward to the point of desired execution. Some simulators, debuggers, or tracing environments (e.g., gdb) provide the ability to fast forward based upon a program PC, and the number of times that PC was executed. We therefore, provide the instruction PC for the start of the simulation point, the procedure that PC occurred in, and the number of times that PC has to be executed in order to arrive at the desired simulation point.

These results show that a single simulation point can be accurate for many programs, but there is still a significant amount of error for programs like `bzip`, `gzip` and `gcc`. This occurs because there are many different phases of execution in these programs, and a single simulation point will not accurately represent all of the different phases. To address this, we used our clustering analysis to find multiple simulation points to accurately capture these programs behavior, which we describe next.

5.2 Multiple Simulation Points

To support multiple simulation points, the simulator can be run from start to stop, only performing detailed simulation on the selected intervals. Or the simulation can be broken down into N simulations, where N is the number of clusters found via analysis, and each simulation is run separately. This has the further benefit of breaking the simulation down into parallel components that can be distributed across many processors. This is the methodology we use in our simulator. For both cases results from the separate simulation points need to be weighed and combined to arrive at overall performance for the program [4]. Care must be taken to combine statistics correctly (simply averaging will give incorrect results for statistics such as rates).

Knowing the clustering alone is not sufficient to enable multiple point simulation because the cluster centers do not correspond to actual intervals of execution. Instead, we must first pick a representative for each cluster that will be used to approximate the behavior of the the full cluster. In order to pick this representative, we choose for each cluster the actual interval that is closest to the center (centroid) of the cluster. In addition to this, we weigh any use of this representative by the size of the cluster it is representing. If a cluster has only one point, it's representative will only have a small impact on the overall outcome of the program.

Table 2 shows the multiple simulation points found for all of the SPEC 2000 benchmarks. For these results we limited the number of clusters to be at most six for all but the most complex programs. This was done, in order to limit the number of simulation points, which also limits the amount of warmup time needed to perform the overall simulation. The cluster formation algorithm in Section 4 takes as an input parameter the max number of clusters to be allowed. Each simulation point contains two numbers. The first number is the location of the simulation point in 100s of millions of instructions. The second number in parentheses is the weight for that simulation point, which is used to create an overall combined metric. Each simulation point corresponds to 100 million instructions.

Figure 10 shows the IPC results for multiple simulation points. The first bar shows our single simulation points simulating for 100 million instructions. The second bar `LongSP` chooses a single simulation point, but the length of simulation is identical to the length used for multiple simulation points (which may go up to 1 billion instructions). This is to provide a fair comparison between the single simulation points and multiple. The `Multiple` bar shows results using the multiple simulation points, and the final bar is IPC for full simulation. As in Figure 9, the closer the bar is to `Full`, the better.

The results show that the average IPC error rate is reduced to 3% using multiple simulation points, which is down from 17% using the long single simulation point. This is significantly lower than the average 80% error seen for blindly fast forwarding. The benefits can be most clearly seen in the programs `bzip`, `gcc`, `ammp`, and `galgel`. The reason that the long contiguous simulation points do not do much better is that they are constrained to only sample at one place in the program. For many programs this is sufficient, but for those with interesting long term behavior, such as `bzip`, it is

name	Len	Init	SP	PC	Proc Name	Multiple SimPoints				
ammp	3265	24	109	026834	mm_fv_update.	3026(13.8) 1607(12.6)	1774(31) 2437(4.9)	595(15.3) 3112(11.5)	1068(1.3) 2480(2.2)	2128(7.4)
applu	2238	3	2180	018520	buts_	524(22.1) 1507(14.5)	1625(22.5)	1956(18.8)	2234(6.6)	1380(15.5)
apsi	3479	3	3409	0380ac	dctdxf_	2107(5.6)	2863(14)	1007(70.7)	896(7.7)	1618(2)
art-110	417	75	341	00fbb0	match	82(42.9)	255(41.2)	50(15.8)		
art-470	450	83	366	00f5d0	match	300(36.2)	46(14.7)	236(49.1)		
bzip2-graphic	1435	4	719	012a5c	spec_putc	168(11.7) 519(11.6)	1042(3.7) 872(8.2)	430(7.5) 195(5.6)	762(16.2) 148(2)	106(15.3) 1435(18.2)
bzip2-program	1249	4	459	00ddd0	sortIt	140(11) 1005(7)	468(12.3) 94(6.9)	78(6.2) 606(14)	990(16) 859(14.6)	445(7.4) 341(4.7)
bzip2-source	1088	4	978	00d774	qSort3	395(16) 177(34.7)	511(4.3)	64(29.1)	488(7.3)	530(8.6)
crafty	1918	462	775	021730	SwapXray	123(25)	510(19.7)	664(22.7)	1123(32.5)	
eon-rushmeier	578	140	404	04e1b4	viewingHit	260(6.6)	238(23.7)	337(20.9)	435(35.6)	216(13.1)
equake	1315	35	813	012410	phi0	874(12.2) 62(11.6)	1292(36.7)	463(12.2)	336(24.1)	3(3.2)
facerec	2682	356	376	02d1f4	graphroutines_lo.	1976(60.1)	1528(2.5)	1935(3.9)	1398(29.2)	348(4.3)
fma3d	2683	192	2542	0e3140	scatter_element.	112(7) 509(13)	209(0.6)	842(68.4)	1600(11)	47(0.1)
galgel	4093	3	2492	02db00	syshtn_	3511(5.5) 2181(29)	2081(11) 2161(3.3)	3466(11.2) 1017(5.5)	516(31.6)	2141(2.7)
gap	2695	639	675	050750	CollectGarb	1114(8.2)	1196(58.1)	88(12.7)	2189(14)	2609(7.1)
gcc-166	469	61	390	0d157c	gen_rtx	238(6.4)	149(42.2)	30(21.3)	404(30.1)	
gcc-200	1086	151	737	0ceb04	refers_to_regno.	8(45.8)	587(17.9)	921(10.9)	575(14.5)	1011(11)
gcc-expr	120	27	37	191fd0	validate_change	63(12.5) 88(5)	81(15.8)	42(16.7)	25(4.2)	9(45.8)
gcc-integrate	131	14	5	1198e0	find_single_use.	118(9.2) 73(17.6)	41(27.5)	102(21.4)	9(20.6)	57(3.8)
gcc-scilab	620	139	208	100d54	insert	255(54.2)	39(9.5)	231(13.2)	379(15.8)	170(7.3)
gzip-graphic	1037	158	654	009c00	fill_window	961(45.4) 566(13.4)	87(28.5)	373(7.3)	1(0.1)	461(5.2)
gzip-log	395	91	266	00d280	inflate_codes	207(24.1)	171(16.5)	157(16.7)	330(23.5)	71(19.2)
gzip-program	1688	112	1190	009660	longest_match	228(22.7)	779(21.4)	472(9.1)	1410(20.4)	594(26.4)
gzip-random	821	152	624	00a14c	deflate	484(0.9) 1(0.1)	625(0.2)	580(51)	811(16.8)	200(30.9)
gzip-source	843	68	335	00a224	deflate	248(14.5) 720(2.5)	327(13.2)	167(17.7)	656(27.8)	373(24.4)
lucas	1423	11	546	021ef0	fft_square_	982(21.4)	602(10.7)	1370(21.4)	468(28)	524(18.6)
mcf	618	15	554	00911c	price_out_impl k	268(39.6) 143(3.9)	425(11)	205(30.1)	468(4.5)	316(10.8)
mesa	2816	6	1136	0a30f0	general_textured.	1846(35.3)	2806(0.7)	398(35.3)	977(28.8)	
mgrid	4191	21	3293	0160f0	resid_	43(24.2)	3459(22.8)	807(20.1)	3110(16.3)	2476(16.6)
parser	5467	388	1147	01edfc	region_valid	3342(25.1)	1771(29.8)	5102(19.7)	2008(19.4)	4772(6)
perlbmk-diff	399	56	142	07f974	regmatch	6(1) 239(31.8)	355(62.7)	11(0.5)	397(0.8)	12(3.3)
perlbmk-make	20	3	12	08268c	Perl_runops_st.	1(5)	20(20)	6(75)		
perlbmk-perf	290	69	6	08268c	Perl_runops_st.	39(59.3)	207(40.7)			
perlbmk-split	1108	162	451	07fc98	regmatch	704(44.9)	596(9.1)	232(21.7)	461(21.8)	501(2.6)
sixtrack	4709	250	3044	167894	thin6d_	6(1.7)	1719(98.3)			
swim	2258	3	2080	019130	calc1_	1951(29.8)	38(14)	777(24.7)	710(13.8)	2101(17.8)
twolf	3464	7	1067	041094	ucxxl	312(17)	2888(11.3)	3268(11.7)	961(20.4)	2054(39.5)
vortex-one	1189	36	272	06289c	Mem_GetWord	536(17.1)	366(23.3)	115(8.2)	1058(17.2)	878(34.2)
vortex-three	1330	177	565	0336a8	Part_Delete	934(25.4) 485(25.4)	1129(11.4)	96(8.9)	47(11.1)	586(17.8)
vortex-two	1386	206	1025	05e6fc	Mem_NewRegion	635(7.6) 397(23.2)	752(24.5)	554(21.9)	930(7.4)	360(15.3)
vpr-place	1122	4	593	0224ec	get_non_update.	166(25.5) 547(27.9)	857(21.6)	1(0.2)	362(12.8)	1057(12)
vpr-route	840	12	477	025c80	get_heap_head	559(29.9)	89(28)	353(23.8)	3(2.6)	490(15.7)
wupwise	3496	11	3238	01d680	zgemm_	1811(43.3)	91(8)	3055(43.2)	1524(5.4)	

Table 2: Single simulation points for SPEC 2000 benchmarks. Columns 2 through 4 are in terms of 100 million instruction executed. The length of full execution is shown, as well as the end of initialization. SP is the single simulation point using the approach in this paper. The procedure in which the simulation point occurred and its PC are also shown. The last 6 digits of PC of each SimPoint is given in hex, so the address is formed from 120xxxxxx. Procedure names that end in "." were truncated due to space. The rest of the columns list the multiple simulation points found in 100s of millions. The first number is the starting place of the simulation point relative to the *start* of execution. The second number shows the weight given to the cluster that simulation point was taken from, and is used when weighing the final results of the simulation.

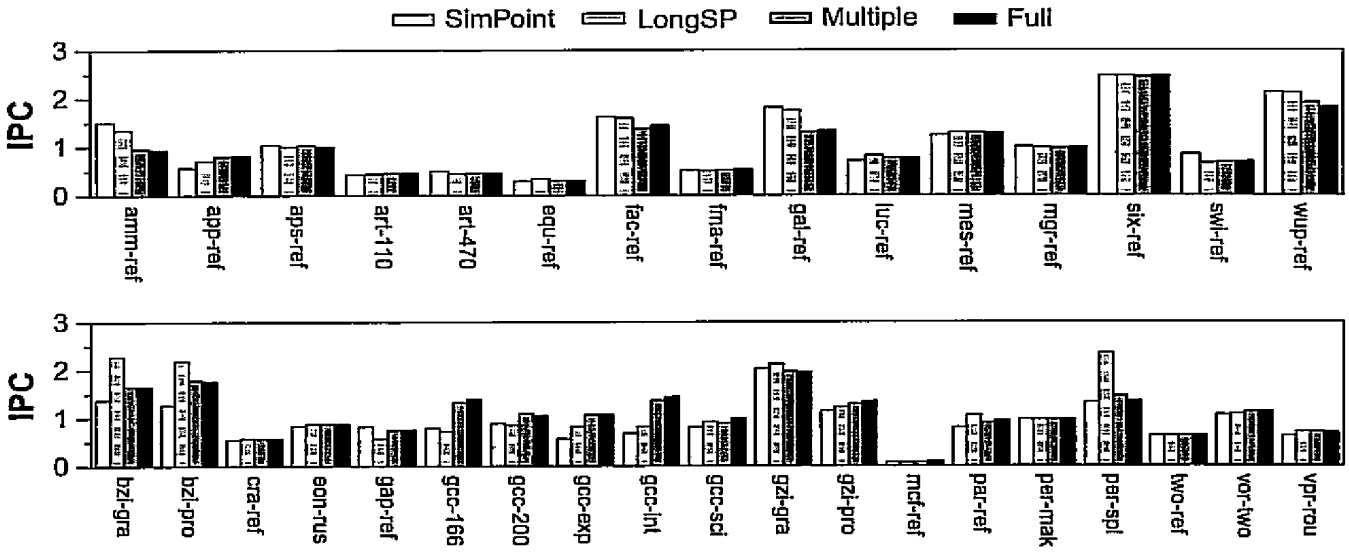

Figure 10: Multiple simulation point results. Simulation results are shown for using a single simulation point simulating for 100 million instructions, LongSP chooses a single simulation point simulating for the same length of execution as the multiple point simulation, simulation using multiple simulation points, and the full execution of the program.

impossible to approximate the full behavior.

Figure 11 is the average over all of the floating point programs (top graph) and integer programs (bottom graph). Errors for IPC, branch miss rate, instruction and data cache miss rates, and the unified L2 cache miss rate for the architecture presented in Section 2 are shown. The errors are with respect to these metrics for the full length of simulation using SimpleScalar. Results are shown for starting simulation at the start of the program `None`, blindly fast forwarding a billion instructions `FF-Billion`, single simulation points of duration 1 (`SimPoint`) and k (`LongSP`), and multiple simulation points (`Multiple`).

The first thing to note is that using the just a single small simulation point performs quite well on average across all of the metrics when compared to blindly fast-forwarding. Even though a single SimPoint does well, it is clearly beaten by using the clustering based scheme presented in this paper across all of the metrics examined. One thing that stands out on the graphs is that the error rate of the instruction cache and L2 cache appear to be high (especially for the integer programs) despite the fact that our technique is doing quite well in terms of overall performance. This is due to the fact that we present here an arithmetic mean of the errors, and there are several programs that have high error rates due to the very small number of cache misses. If there are 10 misses in the whole program, and we estimate there to be 100, that will result in a error of 10X. We point to the overall IPC as the most important metric for evaluation as it implicitly weighs each of the metrics by it's relative importance.

6. RELATED WORK

Time Varying Behavior of Programs: In [18], we provided a first attempt at showing the periodic patterns for all of the SPEC 95 programs, and how these vary over time for cache behavior, branch prediction, value prediction, address prediction, IPC and RUU occupancy.

Training Inputs and Finding Smaller Representative Inputs: One approach for reducing the simulation time is to use the training or test inputs from the SPEC benchmark suite. For many of the benchmarks, these inputs are either (1) still too long to fully simulate, or (2) too short and place too much emphasis on the startup and shutdown parts of the program's execution, or (3) inaccurately estimate behavior such as cache accesses do to decreased working set size.

KleinOsowski et. al [12], have developed a technique where they manually reduce the input sets of programs. The input sets were developed using a range of approaches from truncating of the input files to modification of source code to reduce the number of times frequent loops were traversed. For these input sets they develop, they make sure that they have similar results in terms of IPC, cache, and instruction mix.

Fast Forwarding and Check-pointing: Historically researchers have simulated from the start of the application, but this usually does not represent the majority of the program's behavior because it is still in the initialization phase. Recently researchers have started to *fast-forward* to a given point in execution, and then start their simulation from there, ideally skipping over the initialization code to an area of code representative of the whole. During fast-forward the simulator simply needs to act as a functional simulator, and may take full advantage of optimizations like direct execution. After the fast-forward point has been reached, the simulator switches to full cycle level simulation.

After fast-forwarding, the architecture state to be simulated is still cold, and a warmup time is needed in order to start collecting representative results. Efficiently warming up execution only requires references immediately proceeding the start of simulation. Haskins and Skadron [7] examined probabilistically determining the minimum set of fast-forward transactions that must be executed for warm up to accurately produce state as it would have appeared had the entire fast-forward interval been used for warm up [7]. They

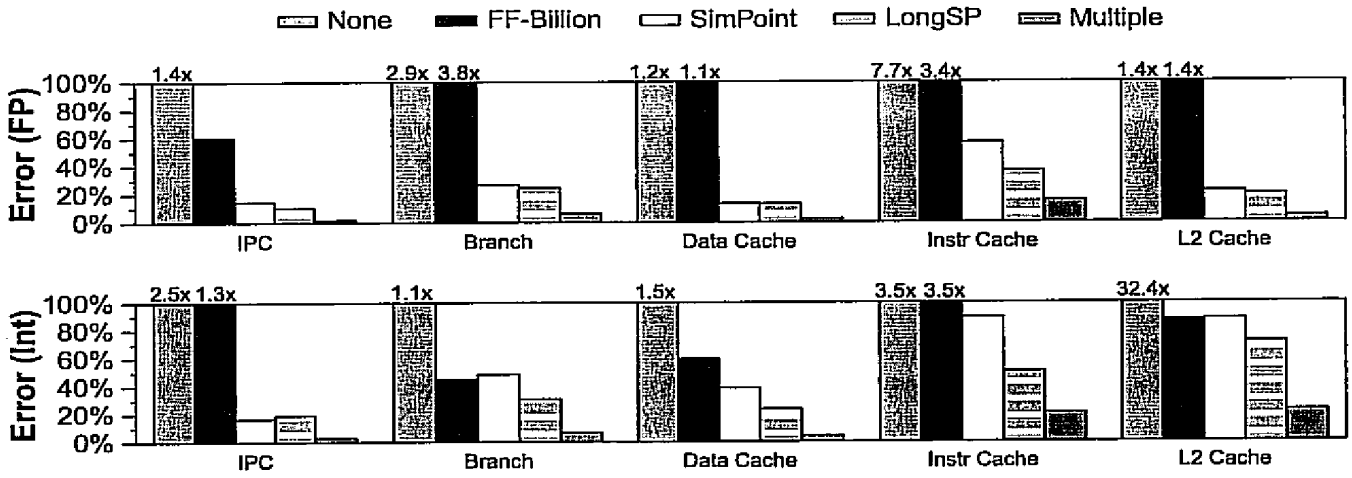

Figure 11: Average error results for the SPEC 2000 floating point (top) and integer (bottom) benchmarks for IPC, branch misprediction, instruction, data and unified L2 cache miss rates.

recently examined using reuse analysis to determine how far before full simulation warmup needs to occur [8].

An alternative to fast forwarding is to use check-pointing to start the simulation of a program at a specific point. With check-pointing, code is executed to a given point in the program and the state is saved, or checkpointed, so that other simulation runs can start there. In this way the initialization section can be run just one time, and there is no need to fast forward past it each time. The architectural state (e.g., caches, register file, branch prediction, etc) can either be stored in the trace (if they are not going to change across simulation runs) or can be warmed up in a manner similar to described above.

Automatically Finding Where to Simulate: Our work is based upon the basic block distribution analysis in [19] as described in prior sections. Recent work on finding simulation points for data cache simulations is presented by Lafage and Seznec [13]. They proposed a technique to gather statistics over the complete execution of the program and use them to choose a representative slice of the program. They evaluate two metrics, one which captures memory spatial locality and one which captures memory temporal locality. They further propose to create specialized metrics such as instruction mix, control transfer, instruction characterization, and distribution of data dependency distances to further quantify the behavior of the both the program's full execution and the execution of samples.

Statistical Sampling: Several different techniques have been proposed for sampling to estimate the behavior of the program as a whole. These techniques take a number of contiguous execution samples, referred to as clusters in [4], across the whole execution of the program. These clusters are spread out throughout the execution of the program in an attempt to provide a representative section of the application being simulated. Conte et. al [4] formed multiple simulation points by randomly picking intervals of execution, and then examining how these fit to the overall execution of the program for several architecture metrics (IPC and branch and data cache statistics). Our work is complementary to this, where we provide a fast and metric independent approach for picking multiple simulation points based just on basic block vector similarity. When an architect gets a new binary to examine they can use our approach to quickly find the simulation points, and then validate these with detailed simulation in parallel with using the binary.

Statistical Simulation: Another technique to improve simulation time is to use statistical simulation [16]. Using statistical simulation, the application is run once and a synthetic trace is generated that attempts to capture the whole program behavior. The trace captures such characteristics as basic block size, typical register dependencies and cache misses. This trace is then run for sometimes as little as 50-100,000 cycles on a much faster simulator. Nussbaum and Smith [15] also examined generating synthetic traces and using these for simulation and was proposed for fast design space exploration. We believe the techniques presented in this paper are complementary to the techniques of Oskin et al. and Nussbaum and Smith in that more accurate profiles can be determined using our techniques, and instead of attempting to characterize the program as a whole it can be characterized on a per-phase basis.

7. SUMMARY

At the heart of computer architecture and program optimization is the need for understanding program behavior. As we have shown, many programs have wildly different behavior on even the very largest of scales (over the full lifetime of the program). While these changes in behavior are drastic, they are not without order, even in very complex applications such as gcc. In order to help future compiler and architecture researchers in exploiting this large scale behavior, we have developed a set of analytical tools that are capable of automatically and efficiently analyzing program behavior over large sections of execution.

The development of the analysis is founded on a hardware independent metric, *Basic Block Vectors*, that can concisely summarize the behavior of an arbitrary section of execution in a program. We showed that by using Basic Block Vectors one can capture the behavior of programs as defined by several architectural metrics (such as IPC, and branch and cache miss rates).

Using this framework, we examine the large scale behavior of several complex programs like gzip, bzip, and gcc, and find interesting patterns in their execution over time. The

behavior that we find shows that code and program behavior repeat over time. For example, in the input we examined in detail for gcc we see that program behavior repeats itself every 23.6 billion instructions. Developing techniques that automatically capture behavior on this scale is useful for architectural, system level, and runtime optimizations. We present an algorithm based on the identification of clusters of basic block vectors that can find these repeating program behaviors and group them into sets for further analysis. For two of the programs gzip and gcc we show how the clustering algorithm results line up nicely with the similarity matrix and correlate with the time varying IPC and data cache miss rates.

It is increasingly common for computer architects and compiler designers to use a small section of a benchmark to represent the whole program during the design and evaluation of a system. This leads to the problem of finding sections of the program's execution that will accurately represent the behavior of the full program. We show how our clustering analysis can be used to automatically find multiple simulation points to reduce simulation time and to accurately model full program behavior. We call this clustering tool to find single and multiple simulation points *SimPoint*. SimPoint along with additional simulation point data can be found at: `http://www.cs.ucsd.edu/~calder/simpoint/`. For the SPEC 2000 programs, we found that starting simulation at the start of the program results in an average error of 210% when compared to the full simulation of the program, whereas blindly fast forwarding resulted in an average 80% IPC error. Using a single simulation point found, using our basic block vector analysis, resulted in an average 17% IPC error. When using the clustering algorithm to create multiple simulation points we saw an average IPC error of 3%.

Automatically identifying the phase behavior using clustering is beneficial for architecture, compiler, and operating system optimizations. To this end, we have used the notion of basic block vectors and a random projection to create an efficient technique for identifying phases on-the-fly [20], which can be efficiently implemented in hardware or software. Besides identifying phases, this approach can predict not only when a phase change is about to occur, but to which phase it is about to transition. We believe that using phase information can lead to new compiler optimizations with code tailored to different phases of execution, multi-threaded architecture scheduling, power management, and other resource distribution problems controlled by software, hardware or the operating system.

Acknowledgments

We would like to thank Suleyman Sair and Chris Weaver for their assistance with SimpleScalar, as well as Mark Oskin and the anonymous reviewers for providing helpful comments on this paper. This work was funded in part by DARPA/ITO under contract number DABT63-98-C-0045 and NSF CAREER grant No. CCR-9733278.

8. REFERENCES

[1] A. Ben-Dor, R. Shamir, and Z. Yakhini. Clustering gene expression patterns. *Journal of Computational Biology*, 6:281–297, 1999.

[2] C. M. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford, 1995.

[3] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[4] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the 1996 International Conference on Computer Design (ICCD)*, October 1996.

[5] S. Dasgupta. Experiments with random projection. In *Uncertainty in Artificial Intelligence: Proceedings of the Sixteenth Conference (UAI-2000)*, pages 143–151, San Francisco, CA, 2000. Morgan Kaufmann Publishers.

[6] G. Hamerly and C. Elkan. Learning the k in k-means. Technical Report CS2002-0716, University of California, San Diego, 2002.

[7] J. Haskins and K. Skadron. Minimal subset evaluation: Rapid warm-up for simulated hardware state. In *Proceedings of the 2001 International Conference on Computer Design*, September 2001.

[8] J. Haskins and K. Skadron. Memory reference reuse latency: Accelerating sampled microarchitecture simulations. Technical Report CS-2002-19, U of Virginia, July 2002.

[9] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[10] J.-M. Jolion, P. Meer, and S. Bataouche. Robust clustering with applications in computer vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(8):791–802, 1991.

[11] R. E. Kass and L. Wasserman. A reference Bayesian test for nested hypotheses and its relationship to the schwarz criterion. *Journal of the American Statistical Association*, 90(431):928–934, 1995.

[12] A. KleinOsowski, J. Flynn, N. Meares, and D. Lilja. Adapting the spec 2000 benchmark suite for simulation-based computer architecture research. In *Proceedings of the International Conference on Computer Design*, September 2000.

[13] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. In *Workload Characterization of Emerging Applications, Kluwer Academic Publishers*, September 2000.

[14] J. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. LeCam and J. Neyman, editors, *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, Berkeley, CA, 1967. University of California Press.

[15] S. Nussbaum and J. E. Smith. Modeling superscalar processors via statistical simulation. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[16] M. Oskin, F. T. Chong, and M. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor designs. In *27th Annual International Symposium on Computer Architecture*, June 2000.

[17] D. Pelleg and A. Moore. X-means: Extending K-means with efficient estimation of the number of clusters. In *Proceedings of the 17th International Conf. on Machine Learning*, pages 727–734. Morgan Kaufmann, San Francisco, CA, 2000.

[18] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report UCSD-CS99-630, UC San Diego, August 1999.

[19] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[20] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. Technical Report CS2002-0710, UC San Diego, June 2002.

[21] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[22] O. Zamir and O. Etzioni. Web document clustering: A feasibility demonstration. In *Research and Development in Information Retrieval*, pages 46–54, 1998.

EXHIBIT 12
TO DECLARATION OF
JAMES A FLIGHT

INTERNET ARCHIVE
WayBackMachine

Enter Web Address: http:// All Take Me Back Adv. Search

Searched for http://www-cse.ucsd.edu/Dienst/UI/2.0/Describe/ncstrl.ucsd_cse/CS2002-0710 **3** Results

* denotes when site was updated.

Search Results for Jan 01, 1996 - Jul 06, 2007

1996	1997	1998	1999	2000	2001	2002	2003	2004	2005	2006	2007
0 pages	0 pages	0 pages	0 pages	0 pages	0 pages	1 pages	1 pages	0 pages	0 pages	0 pages	1 pages
						Nov 19, 2002 *	Jul 09, 2003				Feb 06, 2007

EXHIBIT 13
TO DECLARATION OF
JAMES A FLIGHT

Phase Tracking and Prediction

Timothy Sherwood, Suleyman Sair and Brad Calder
CS2002-0710
June 23, 2002

In a single second a modern processor can execute billions of instructions. Obtaining a bird's eye view of the behavior of a program at these speeds can be a difficult task when all that is available is cycle by cycle examination. In many programs, behavior is anything but steady state, and understanding the patterns of behavior at run-time can unlock a multitude of optimization opportunities. In this paper we present a unified profiling architecture that can efficiently capture, classify, and predict program behavior on the largest of time scales all at run-time with no support from software. By examining the proportion of instructions that were executed from different sections of code, we can find generic phases that correspond to changes in behavior across many metrics. By classifying phases generically, we avoid the need to identify phases for each optimization, and enable a unified prediction scheme that can forecast future behavior. We examine the ability of our phase tracking architecture to accurately capture the phase behavior of a program's execution with respect to its overall performance (IPC), branch prediction, cache performance, and energy, and show how phase behavior may be captured efficiently using a simple predictor.

How to view this document

- Display the **whole** document in one of the following formats.
  - PostScript 10012 bytes.
- Print or download all or selected pages.

EXHIBIT 14
TO DECLARATION OF
JAMES A FLIGHT

Phase Tracking and Prediction

Timothy Sherwood, Suleyman Sair and Brad Calder
CS2002-0710
June 23, 2002

In a single second a modern processor can execute billions of instructions. Obtaining a bird's eye view of the behavior of a program at these speeds can be a difficult task when all that is available is cycle by cycle examination. In many programs, behavior is anything but steady state, and understanding the patterns of behavior at run-time can unlock a multitude of optimization opportunities. In this paper we present a unified profiling architecture that can efficiently capture, classify, and predict program behavior on the largest of time scales all at run-time with no support from software. By examining the proportion of instructions that were executed from different sections of code, we can find generic phases that correspond to changes in behavior across many metrics. By classifying phases generically, we avoid the need to identify phases for each optimization, and enable a unified prediction scheme that can forecast future behavior. We examine the ability of our phase tracking architecture to accurately capture the phase behavior of a program's execution with respect to its overall performance (IPC), branch prediction, cache performance, and energy, and show how phase behavior may be captured efficiently using a simple predictor.

How to view this document

- Display the **whole** document in one of the following formats.
  - PostScript 10012 bytes.
- Print or download all or selected pages.

[Search]

EXHIBIT 15
TO DECLARATION OF
JAMES A FLIGHT

James A. Flight

From: James A. Flight
Sent: Saturday, July 07, 2007 1:45 PM
To: 'calder@cs.ucsd.edu'
Subject: Request for citation assistance
Attachments: CS2002-0710.ps; CS2002-0710 (2).pdf
Signed By: jflight@hfzlaw.com

Sensitivity: Confidential

Brad,

We are trying to determine when the full text of your article "Phase tracking and Prediction" was available to the public. On its face, it appears to have been published in June of 2003. However, this link (http://www.cse.ucsd.edu/Dienst/UI/2.0/Describe/ncstrl.ucsd_cse/CS2002-0710) also available through here (http://www.cs.ucsd.edu/Dienst/UI/2.0/ListAuthors/S?authority=ncstrl.ucsd_cse) makes it appear that the article may have been available on request to you as early as June of 2002 (see the postscript file attached as a PDF hereto for your convenience). However, following the link in the Postscript file leads to this list (http://www-cse.ucsd.edu/users/calder/papers.html), which identifies the 2003 publication, and no 2002 publication.

In view of the foregoing, can you help me understand what, if anything, was available in 2002? I suspect it was merely the abstract, but perhaps you had the draft ready and were making it available a year before its official 2003 publication?

Thanks, in advance, for any assistance you are able to provide.

Best regards,

Jim

James A. Flight
HFZ HANLEY FLIGHT& ZIMMERMAN...

150 South Wacker Drive, Suite 2100
Chicago, Illinois 60606

(312) 580-1034 (Direct)
(312) 580-1020 (Main)
(312) 580-9696 (Fax)

jflight@hfzlaw.com

Techreport

To obtain a copy of this techreport, please

look for it at the following site:

http://www-cse.ucsd.edu/users/calder/papers.html

Or send email or a letter to:

Brad Calder

University of California, San Diego

9500 Gilman Drive

La Jolla, CA 92093-0114

calder@cs.ucsd.edu

EXHIBIT 16
TO DECLARATION OF
JAMES A FLIGHT

James A. Flight

From: James A. Flight
Sent: Monday, July 16, 2007 11:42 AM
To: 'calder@cs.ucsd.edu'
Subject: RE: Request for citation assistance
Signed By: jflight@hfzlaw.com

Sensitivity: Confidential

Brad,

I was wondering if you have had an opportunity to consider this issue. I would appreciate any assistance you are able to provide.

Thank you

Jim

From: James A. Flight
Sent: Saturday, July 07, 2007 1:45 PM
To: 'calder@cs.ucsd.edu'
Subject: Request for citation assistance
Sensitivity: Confidential

Brad,

We are trying to determine when the full text of your article "Phase tracking and Prediction" was available to the public. On its face, it appears to have been published in June of 2003. However, this link (http://www.cse.ucsd.edu/Dienst/UI/2.0/Describe/ncstrl.ucsd_cse/CS2002-0710) also available through here (http://www.cs.ucsd.edu/Dienst/UI/2.0/ListAuthors/S?authority=ncstrl.ucsd_cse) makes it appear that the article may have been available on request to you as early as June of 2002 (see the postscript file attached as a PDF hereto for your convenience). However, following the link in the Postscript file leads to this list (http://www-cse.ucsd.edu/users/calder/papers.html), which identifies the 2003 publication, and no 2002 publication.

In view of the foregoing, can you help me understand what, if anything, was available in 2002? I suspect it was merely the abstract, but perhaps you had the draft ready and were making it available a year before its official 2003 publication?

Thanks, in advance, for any assistance you are able to provide.

Best regards,

Jim

James A. Flight
HFZ HANLEY FLIGHT& ZIMMERMAN...

150 South Wacker Drive, Suite 2100
Chicago, Illinois 60606

(312) 580-1034 (Direct)
(312) 580-1020 (Main)
(312) 580-9696 (Fax)

jflight@hfzlaw.com

EXHIBIT 17
TO DECLARATION OF
JAMES A FLIGHT

James A. Flight

From: postmaster@hfzlaw.com
Sent: Monday, July 16, 2007 11:45 AM
To: James A. Flight
Subject: Delivery Status Notification (Relay)
Attachments: ATT802303.txt; RE: Request for citation assistance

This is an automatically generated Delivery Status Notification.

Your message has been successfully relayed to the following recipients, but the requested delivery status notifications may not be generated by the destination.

calder@cs.ucsd.edu

EXHIBIT 18
TO DECLARATION OF
JAMES A FLIGHT

James A. Flight

From: CSE Computing Support [webmaster@cs.ucsd.edu]
Sent: Friday, July 06, 2007 7:48 PM
To: James A. Flight
Subject: [website #200731]: request for assistance.

We had a couple of problems with the techreport server. I kicked it and it seems to be working now. That techreport is available either by following the link you gave, or by going to

http://www.cs.ucsd.edu/facresearch/technicalreports/techreports.html

From there you can search by Author or by Year and also get to the report.

Let us know if you have any other problems.

-glenn

On Fri, 6 Jul 2007 14:20:22 -0700, JFlight@hfzlaw.com wrote:
> Hello,
>
>
>
> The publication Sherwood et al, Phase Tracking and Prediction, is
> dated June 2003. However, according to the Internet, at least some
> portion of that article was available on your cite
> (www.cse.ucsd.edu/Dienst/UI/2.0/Describe/ncstrl.ucsd_cse/CS2002-0710)
> in June of 2002. According to the Internet Wayback machine, this page
> was on-line as of November 19, 2002. Can you assist my research by
> telling me what exactly was available on your cite in the 202 time
> frame (e.g., by giving me a copy of the postscript file)?
>
>
>
> Thank you,
>
>
>
> Jim
>
>
>
> James A. Flight
>
>
>
>
>
> 150 South Wacker Drive, Suite 2100
> Chicago, Illinois 60606
>
> (312) 580-1034 (Direct)
> (312) 580-1020 (Main)

> (312) 580-9696 (Fax)
>
> jflight@hfzlaw.com
>
>
>
>
>
> Important: This electronic mail message and any attached files contain
> information intended for the exclusive use of the individual or entity
> to whom it is addressed and may contain information that is
> proprietary, privileged, confidential and/or exempt from disclosure
> under applicable law. If you are not the intended recipient, please
> notify the sender, by electronic mail or telephone, of any unintended
> recipients and delete the original message without making any copies.